

Microsoft SQL Server 2022

Microsoft SQL Server 2022

НАИБОЛЕЕ ПОЛНОЕ РУКОВОДСТВО



Александр Бондарь — преподаватель, опытный технический писатель. Автор переводов на русский язык книг Х. Борри «Firebird. Руководство разработчика баз данных» и Д. Петковича «Microsoft SQL Server 2008. Руководство для начинающих», а также автор книг «InterBase и Firebird. Практическое руководство для умных пользователей и начинающих разработчиков», «Microsoft SQL Server 2012» и «Microsoft SQL Server 2014». Александр является прирожденным методистом, умеющим излагать сложные вопросы программирования доступным для читателя языком.

Книга посвящена установке, настройке, администрированию и разработке баз данных с помощью СУБД MS SQL Server 2022. Подробно описана разработка баз данных MS SQL, включая проектирование БД, создание самой БД и всех необходимых для ее эффективного функционирования объектов (таблицы, индексы, хранимые процедуры, триггеры, функции). Рассмотрены основы языка запросов, показаны методы проектирования, создания и изменения таблиц. Подробно изучаются индексы, заполнение таблиц данными, изменение и удаление данных, выборка из базы, представления, транзакции, их характеристики и взаимодействие, хранимые процедуры и триггеры. В книге приведено множество примеров использования различных средств SQL Server.



Примеры из книги можно скачать по ссылке <https://zip.bhv.ru/9785977518055.zip>, а также со страницы книги на сайте bhv.ru.

ISBN 978-5-9775-1805-5



191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru



Материалы
на www.bhv.ru



В ПОДЛИННИКЕ®

Microsoft SQL Server 2022

- Общие сведения о SQL Server 2022
- Принципы работы с базами данных
- Создание базы данных и основных ее объектов
- Проектирование, создание и изменение таблиц
- Заполнение таблиц данными, изменение и удаление данных
- Индексы
- Представления
- Транзакции, их характеристики и взаимодействие
- Выборка данных
- Хранимые процедуры, функции, определенные пользователем, триггеры
- Средства отображения объектов и их характеристик



УДК 004.65
ББК 32.973.26-018.2
Б81

Бондарь А. Г.

Б81 Microsoft SQL Server 2022. — СПб.: БХВ-Петербург, 2024. — 528 с.: ил. —
(В подлиннике)

ISBN 978-5-9775-1805-5

Книга посвящена установке, настройке, администрированию и разработке баз данных с помощью СУБД MS SQL Server 2022. Материал сопровождается большим количеством примеров кода, которые можно использовать на практике. Рассмотрены основы языка запросов, используемые типы данных, создание базы данных и основных ее объектов, средства отображения объектов и всех их характеристик. Показаны методы проектирования, создания и изменения таблиц; рассмотрены индексы, заполнение таблиц данными, изменение и удаление данных, выборка из базы, представления, транзакции, их характеристики и взаимодействие, хранимые процедуры и триггеры. Исходные коды примеров размещены на сайте издательства.

Для программистов

УДК 004.65
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	<i>Павел Шалин</i>
Зав. редакцией	<i>Людмила Гауль</i>
Компьютерная верстка	<i>Натальи Смирновой</i>
Оформление обложки	<i>Зои Канторович</i>

Подписано в печать 03.08.23.
Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 42,57.
Тираж 1000 экз. Заказ № 7367.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.
Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-5-9775-1805-5

© Бондарь А. Г., 2024
© Оформление. ООО "БХВ-Петербург", ООО "БХВ", 2024

Оглавление

Введение.....	9
Содержание книги	11
Дополнительные материалы	13
 ГЛАВА 1. Установка программных средств	15
1.1. Установка MS SQL Server 2022 Developer	15
1.2. Установка SQL Server Management Studio.....	19
 ГЛАВА 2. Общие сведения о SQL Server 2022.....	23
2.1. Реляционные базы данных	23
2.1.1. Таблицы	23
2.1.2. Представления.....	29
2.1.3. Хранимые процедуры и триггеры	30
2.1.4. Пользователи, привилегии и роли базы данных	31
2.1.5. Задание первичных ключей таблиц.....	31
2.1.6. Транзакции	32
2.1.7. 12 правил Кодда	33
2.2. Реализация отношений в реляционной модели.....	34
2.2.1. Отношение "один к одному"	34
2.2.2. Отношение "один ко многим"	35
2.2.3. Отношение "многие ко многим"	35
2.3. Нормализация таблиц	36
2.3.1. Цель нормализации таблиц	36
2.3.2. Первая нормальная форма.....	37
2.3.3. Вторая нормальная форма.....	39
2.3.4. Третья нормальная форма	39
2.3.5. Другие нормальные формы.....	40
2.3.6. Денормализация таблиц	41
2.4. Проектирование баз данных	41
2.5. Язык Transact-SQL	43
2.5.1. Синтаксис	43
2.5.2. Основные сведения о составе языка Transact-SQL	51
Что дальше?.....	53

ГЛАВА 3. Работа с базами данных	55
3.1. Запуск и останов экземпляра сервера	56
3.1.1. Запуск экземпляра сервера.....	56
3.1.2. Останов экземпляра сервера	57
3.2. Что собой представляет база данных в SQL Server	58
3.2.1. Системные базы данных.....	59
3.2.2. Базы данных пользователей.....	61
3.2.3. Некоторые характеристики базы данных	62
3.2.4. Некоторые характеристики файлов базы данных	64
3.3. Получение сведений о базах данных и их файлах в текущем экземпляре сервера	65
3.3.1. Системное представление <i>sys.databases</i>	65
3.3.2. Системное представление <i>sys.master_files</i>	66
3.3.3. Системное представление <i>sys.database_files</i>	68
3.3.4. Системное представление <i>sys.filegroups</i>	69
3.3.5. Другие средства получения сведений об объектах базы данных	69
3.4. Создание и удаление базы данных.....	72
3.4.1. Использование операторов Transact-SQL для создания, отображения и удаления баз данных.....	72
3.4.2. Создание базы данных с использованием диалоговых средств Management Studio	112
3.5. Изменение базы данных.....	116
3.5.1. Изменение базы данных в языке Transact-SQL	116
3.5.2. Изменение базы данных диалоговыми средствами Management Studio	128
3.5.3. Удаление базы данных диалоговыми средствами Management Studio	135
3.6. Создание автономной базы данных	135
3.6.1. Установка допустимости автономных баз данных	136
3.6.2. Создание автономной базы данных и пользователя средствами языка Transact-SQL.....	137
3.6.3. Создание автономной базы данных диалоговыми средствами Management Studio	138
3.6.4. Создание автономного пользователя в Management Studio	139
3.6.5. Соединение с автономной базой данных в Management Studio	140
3.7. Создание мгновенных снимков базы данных	141
3.8. Схемы базы данных	143
3.8.1. Работа со схемами в Transact-SQL	143
3.8.2. Работа со схемами в Management Studio.....	146
3.9. Средства копирования и восстановления баз данных	148
3.9.1. Использование операторов Transact-SQL для копирования/восстановления базы данных.....	148
3.9.2. Использование диалоговых средств Management Studio для копирования/восстановления базы данных	149
3.10. Домашнее задание	153
Что дальше?	154

ГЛАВА 4. Типы данных	155
4.1. Классификация типов данных в SQL Server.....	156
4.2. Объявление локальных переменных.....	158
4.3. Числовые типы данных	159
4.3.1. Тип данных <i>BIT</i>	161
4.3.2. Целочисленные типы данных <i>TINYINT, SMALLINT, INT, BIGINT</i>	163
4.3.3. Дробные числа <i>NUMERIC, DECIMAL, SMALLMONEY, MONEY</i>	165
4.3.4. Числа с плавающей точкой <i>FLOAT, REAL</i>	169
4.3.5. Функции для работы с числовыми данными.....	170
4.4. Символьные данные	175
4.4.1. Символьные строки <i>CHAR, VARCHAR</i>	176
4.4.2. Символьные строки <i>NCHAR, NVARCHAR</i>	177
4.4.3. Типы данных <i>VARCHAR(MAX), NVARCHAR(MAX), VARBINARY(MAX)</i>	178
4.4.4. Строковые функции.....	178
4.5. Типы данных даты и времени.....	189
4.5.1. Описание типов данных даты и времени.....	189
4.5.2. Действия с датами и временем	191
4.6. Двоичные данные	201
4.7. Пространственные типы данных	202
4.7.1. Тип данных <i>GEOMETRY</i>	203
4.7.2. Тип данных <i>GEOGRAPHY</i>	215
4.8. Другие типы данных.....	220
4.8.1. Тип данных <i>SQL_VARIANT</i>	220
4.8.2. Тип данных <i>HIERARCHYID</i>	224
4.8.3. Тип данных <i>UNIQUEIDENTIFIER</i>	229
4.8.4. Тип данных <i>CURSOR</i>	230
4.8.5. Тип данных <i>XML</i>	237
4.9. Создание и удаление пользовательских типов данных	249
4.9.1. Синтаксис оператора создания пользовательского типа данных	250
4.9.2. Создание псевдонима средствами Transact-SQL.....	251
4.9.3. Создание псевдонима в диалоговых средствах Management Studio.....	252
4.9.4. Создание пользовательского табличного типа данных средствами Transact-SQL.....	253
4.9.5. Создание пользовательского табличного типа данных диалоговыми средствами Management Studio.....	257
4.9.6. Удаление пользовательского типа данных	257
Что дальше?.....	259
ГЛАВА 5. Работа с таблицами.....	261
5.1. Синтаксис оператора создания таблицы.....	262
5.1.1. Общие характеристики таблицы	263
5.1.2. Определение столбца.....	265

5.1.3. Ограничения столбца и ограничения таблицы	268
5.1.4. Вычисляемые столбцы	274
5.1.5. Набор столбцов	275
5.2. Примеры простых таблиц	276
5.3. Создание секционированных таблиц	289
5.3.1. Синтаксические конструкции	289
5.3.2. Пример создания секционированной таблицы	292
5.3.3. Отображение результатов создания таблицы	301
5.3.4. Изменение характеристик секционированной таблицы	305
5.4. Создание таблиц диалоговыми средствами	307
5.4.1. Создание таблицы секционирования	307
5.4.2. Создание таблицы секционирования, схемы секционирования и функции секционирования	317
5.5. Отображение состояния секционированных таблиц	323
5.6. Файловые потоки	324
5.7. Удаление таблиц	330
5.7.1. Определение зависимостей таблицы	330
5.7.2. Удаление таблицы оператором <i>DROP TABLE</i>	333
5.7.3. Удаление таблицы диалоговыми средствами Manager Studio	333
5.8. Изменение характеристик таблиц	335
5.8.1. Изменение таблиц при использовании оператора Transact-SQL	336
5.8.2. Изменение таблиц средствами Management Studio	341
5.8.3. Построение диаграммы базы данных	366
Что дальше?	368
ГЛАВА 6. Индексы	369
6.1. Отображение индексов	370
6.2. Работа с индексами средствами Transact-SQL	371
6.2.1. Создание обычного (реляционного) индекса	371
6.2.2. Создание индекса для представлений	378
6.2.3. Создание columnstore индекса	379
6.2.4. Создание индекса для столбца XML	380
6.2.5. Создание пространственного индекса	385
6.2.6. Удаление индекса	387
6.2.7. Изменение индекса	388
6.3. Работа с индексами в диалоговых средствах Management Studio	390
6.3.1. Создание индекса в Management Studio	390
6.3.2. Удаление индекса в Management Studio	394
6.3.3. Изменение индекса в Management Studio	394
Что дальше?	394

ГЛАВА 7. Добавление, изменение и удаление данных	395
7.1. Обобщенное табличное выражение	395
7.2. Добавление данных (оператор <i>INSERT</i>)	396
7.3. Изменение данных (оператор <i>UPDATE</i>)	403
7.4. Удаление данных (оператор <i>DELETE</i>)	407
7.5. Удаление строк таблицы (оператор <i>TRUNCATE TABLE</i>)	408
7.6. Добавление, изменение или удаление строк таблицы (оператор <i>MERGE</i>)	408
Что дальше?	414
ГЛАВА 8. Выборка данных	415
8.1. Оператор <i>SELECT</i>	415
8.2. Оператор <i>UNION</i>	424
8.3. Операторы <i>EXCEPT</i> , <i>INTERSECT</i>	424
8.4. Примеры выборки данных	425
8.4.1. Список выбора	425
8.4.2. Упорядочение результата (<i>ORDER BY</i>)	427
8.4.3. Условие выборки данных (<i>WHERE</i>)	428
8.4.4. Соединение таблиц	436
8.4.5. Группировка результатов выборки (<i>GROUP BY</i> , <i>HAVING</i>)	444
8.5. Использование операторов <i>UNION</i> , <i>EXCEPT</i> , <i>INTERSECT</i>	449
Что дальше?	450
ГЛАВА 9. Представления	451
9.1. Синтаксис операторов для представлений	452
9.1.1. Создание представления	452
9.1.2. Изменение представления	453
9.1.3. Удаление представления	453
9.2. Создание представлений в Transact-SQL	454
9.3. Создание представлений диалоговыми средствами Management Studio	458
Что дальше?	460
ГЛАВА 10. Транзакции	461
10.1. Понятие и характеристики транзакций	461
10.2. Операторы работы с транзакциями	462
10.3. Уровни изоляции транзакции	464
Что дальше?	466
ГЛАВА 11. Хранимые процедуры, функции, определенные пользователем, триггеры	467
11.1. Язык хранимых процедур и триггеров	467
11.1.1. Блок операторов <i>BEGIN/END</i>	468

11.2. Хранимые процедуры.....	472
11.2.1. Создание хранимой процедуры	472
11.2.2. Изменение хранимой процедуры.....	474
11.2.3. Удаление хранимой процедуры.....	474
11.2.4. Использование хранимых процедур.....	475
11.3. Функции, определенные пользователем.....	480
11.3.1. Создание функции	480
11.3.2. Изменение функций.....	481
11.3.3. Удаление функций.....	482
11.3.4. Использование функций.....	482
11.4. Триггеры.....	483
11.4.1. Создание триггеров.....	483
11.4.2. Изменение триггеров	485
11.4.3. Удаление триггеров	486
11.4.4. Использование триггеров.....	487
ПРИЛОЖЕНИЯ	491
ПРИЛОЖЕНИЕ 1. Двенадцать правил Кодда	493
ПРИЛОЖЕНИЕ 2. Зарезервированные слова Transact-SQL	495
ПРИЛОЖЕНИЕ 3. Утилита командной строки <i>sqlcmd</i>	501
ПРИЛОЖЕНИЕ 4. Характеристики базы данных	503
П4.1. Параметры <i>Auto</i>	505
П4.2. Параметры доступности базы данных	506
П4.3. Параметры автономной базы данных	508
П4.4. Параметры восстановления.....	509
П4.5. Общие параметры SQL.....	509
П4.6. Параметры компонента Service Broker	513
ПРИЛОЖЕНИЕ 5. Языки, представленные в SQL Server	515
ПРИЛОЖЕНИЕ 6. Описание электронного архива.....	517
Предметный указатель	519

Введение

MS SQL Server версии 2022 (да и многие предыдущие версии) — весьма сложная система, имеющая огромные возможности. Большое число программных компонентов, представлений просмотра каталогов, системных процедур, функций и других средств может сбить с толку. Получить нужный вам результат можно множеством способов, различными путями, используя разные средства, существующие в системе. Я покажу те способы, которыми можно эффективно и без лишних затрат времени и интеллекта пользоваться для достижения конкретного результата. В основном это те средства, которыми пользуюсь лично я или более достойные люди, очень хорошие специалисты в данной области.

Эта книга в первую очередь предназначена тем, кто никогда не работал ни с какой версией MS SQL Server, а может быть, даже и вообще ни с какой системой управления базами данных (СУБД) — ни с реляционной, ни с сетевой, ни даже с иерархической, не говоря уж и о совсем простеньких ("настольных") системах управления данными. Здесь вы не найдете сравнений настоящей версии сервера с предыдущими, детальных описаний того, что нового появилось в SQL Server 2022.

Я главным образом ориентируюсь на разработчиков баз данных (БД) и создателей программного продукта, требующего использования для своей работы баз данных, или на людей, которые собираются стать такими разработчиками и/или программистами. Следует уточнить ту работу, которую в первую очередь выполняют такие специалисты, и на которых в основном ориентировано содержание этой книги.

Это разработка БД и создание программ, использующих базы данных, включая *проектирование БД, создание самой БД и всех необходимых для ее эффективного функционирования 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 применяют в своей работе БД SQL Server, я также постарался 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 СУБД. Здесь в соответствующем порядке описываются действия, выполняемые человеком, которому нужно спроектировать, создать базу данных, затем

заполнить ее своими данными, изменять, удалять данные и, наконец, отыскивать нужные ему данные. Следовательно, и в книге эти действия описываются в том же порядке: создание БД, создание таблиц, добавление, изменение, удаление, выборка данных. Это начало книги, затем идут описания других необходимых действий по работе с БД и соответствующие средства, представленные в системе, которых, надо сказать, огромное множество. Разумеется, все описать в одной книге невозможно. Я старался дать сведения, необходимые и достаточные для вашей эффективной работы в очень большом диапазоне задач, которые нам с необыкновенной щедростью поставляют 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 и стать сертифицированным специалистом корпорации Microsoft в области работы с SQL Server, и именно SQL Server 2022. Однако структура книги отличается от структуры учебных материалов, которые предлагаются специалистами Microsoft. Надеюсь, в лучшую сторону.

Часто при описании достоинств какой-либо программистской литературы 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 она была действительно полезна читателю, у того должен быть определенный запас знаний и умений, имеющих прямое отношение к программированию, к обработке данных. На начальном этапе знакомства с подобными сложными системами мог бы порекомендовать прочитать любую подходящую книгу из серий "step-by-step" (шаг за шагом) или "for dummies" ("для чайников", а точнее "для дебилов").

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, минуя чтение 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. Поэтому я все-таки взял на себя смелость в *главе 2* чуть более подробно описать основные моменты, связанные с системами управления базами данных, и касающиеся только реляционных БД. Теперь с некоторыми основаниями можно произнести такую фразу: "Если вы можете включить компьютер и запустить на выполнение указанную программу, то при желании с блеском освоите материал этой книги". Пожалуй, это не будет слишком большим преувеличением.

Если все эти относительно подробно описанные основы реляционных БД вам хорошо знакомы (то, что все данные представлены в таблицах, наличие нормальных форм, способы нормализации таблиц, средства для описания синтаксиса, назначение языка SQL и др.), не обижайтесь на меня за излишние подробности, а просто пропустите ненужные вам описания и рассуждения. Хотя можете бегло их просмотреть, не тратя много времени.

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

Все действия я выполнял в операционной системе Windows 10.

Новая версия MS SQL Server 2022 может выполняться в ОС Windows 7 и в некоторых других системах.

Подробные сведения о требованиях к операционным системам содержатся на сайте Microsoft.

Первая версия SQL Server была создана в 1989 году. Нынешняя версия 2022 — в ноябре 2022 года.

Содержание книги

♦ Глава 1. Установка программных средств.

В *главе 1* описана установка MS SQL Server 2022 Developer. В этой книге я рассматриваю русифицированную версию системы.

♦ Глава 2. Общие сведения о SQL Server 2022.

Мне заранее неизвестно, насколько человек, читающий эту книгу, знаком с основами реляционных БД, с принятыми средствами описания синтаксиса формальных языков. *Глава 2* посвящена основным понятиям реляционных БД. Здесь очень кратко описано хранение данных в БД, нормальные формы, нормализация таблиц. Рассмотрены способы реализации отношений между данными в реляционных БД (один к одному, один ко многим, многие ко многим). Приведены простые, но взятые из реальной жизни примеры реализации этих отношений, построенные на связке "внешний ключ/первичный (уникальный) ключ".

В этой же главе рассмотрены 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 то SQL Server, Oracle, IBM DB2, InterBase, Firebird, Sybase, PostgreSQL или даже MySQL (простите, если кого-то не упомянул). Здесь же даны описания базовых синтаксических конструкций — это идентификаторы (обычные и с разделителями), числа, строковые константы.

Кратко описаны объекты БД: таблицы, индексы, пользовательские типы данных, представления, хранимые процедуры, триггеры.

♦ Глава 3. Работа с базами данных.

Поскольку разработчику БД 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 БД. Как уже упоминалось, MS SQL Server — весьма сложная система. Сами базы данных в нем имеют множество свойств, характеристик. Так как разработчику БД на начальных этапах своей деятельно-

сти нет острой необходимости вникать во все тонкости и детали организации данных, то в этой главе я не стал описывать базы данных слишком подробно. Однако у того же разработчика может появиться потребность более детально разобраться с некоторыми характеристиками, которые позволят эффективнее задействовать вычислительные ресурсы и смогут повысить производительность системы. Поэтому достаточно подробное описание характеристик БД я поместил в *приложение 4*. В этой главе описываются и файловые группы — в основном создание, изменение, удаление. Эффект использования файловых групп проявляется позже, когда в БД помещаются таблицы, начинается их заполнение и осуществляется выборка данных.

♦ Глава 4. Типы данных.

Тип данных — важнейшее понятие в программировании вообще и в СУБД в частности. В *главе 4* подробно описаны все типы данных SQL Server. Приведены операции над типами данных, допустимые преобразования данных, применяемые функции. Дан синтаксис оператора создания пользовательских типов данных.

♦ Глава 5. Работа с таблицами.

Важнейший объект реляционной базы данных — таблица. В *главе 5* приведен синтаксис оператора создания таблицы. Дано множество примеров создания таблиц из демонстрационной БД BestDatabase, которая на самом деле является упрощенным фрагментом промышленной базы данных. Подробно описано задание столбцов таблицы и задание ограничений для отдельных столбцов и для таблицы в целом.

♦ Глава 6. Индексы.

Рассмотрен синтаксис операторов создания, изменения и удаления индексов. Кластерные индексы. Индексы для представлений. Приведены примеры.

♦ Глава 7. Добавление, изменение и удаление данных.

Описан синтаксис и назначение операторов INSERT, UPDATE, DELETE, TRUNCATE TABLE, MERGE. Даны примеры их использования.

♦ Глава 8. Выборка данных.

Подробно рассмотрен, пожалуй, самый сложный оператор SELECT, позволяющий выбирать данные из одной или более таблиц. Приведены примеры использования средств определения условий выборки данных, группирования результатов, выполнения соединения таблиц (внешних и внутреннего). Также рассмотрены операторы UNION, EXCEPT, INTERSECT.

♦ Глава 9. Представления.

Создание, изменение и удаление представлений. Назначение. Индексированные представления.

♦ Глава 10. Транзакции.

Сформулировано понятие транзакции. Описаны операторы для старта, подтверждения и отмены транзакции, создания контрольных точек. Рассмотрены все уровни изоляции транзакций, существующие в MS SQL Server 2022.

♦ **Глава 11. Хранимые процедуры, функции, определенные пользователем, триггеры.**

Описаны языковые средства Transact-SQL для создания и использования программных компонентов MS SQL Server — хранимых процедур, пользовательских функций и триггеров.

♦ **Приложение 1.**

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

♦ **Приложение 2.**

Дан 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.**

Исключительно для любителей работы с командной строкой здесь кратко описаны параметры утилиты sqlcmd.

♦ **Приложение 4.**

Описано множество характеристик базы данных, которые можно установить при первоначальном создании БД или при ее изменении.

♦ **Приложение 5.**

Описано большое число языков, которые поддерживаются в MS SQL Server, с некоторыми их характеристиками.

♦ **Приложение 6.**

Перечислены скрипты создания демонстрационной БД и заполнение ее данными.

Дополнительные материалы

Скрипты (иногда употребляют термин "сценарий") для создания учебной базы данных, используемой в этой книге, и заполнения ее данными можно скачать с сайта издательства:

<https://zip.bhv.ru/9785977518055.zip>

Имя каждого скрипта начинается с его порядкового номера. Для создания БД BestDatabase, всех ее объектов и для заполнения таблиц данными нужно выполнять скрипты именно в этом порядке.

Инсталляция программных средств

Версия MS SQL Server 2022 вышла в ноябре 2022 года. Теперь система может выполняться не только под Windows, но и под Linux.

Существуют следующие редакции системы для Windows:

- ◆ Enterprise — с полной функциональностью. Имеет возможность работы в облачном сервисе на платформе Azure. Azure Synapse Link для SQL позволяет выполнять аналитику, бизнес-аналитику и машинное обучение с использованием базы данных SQL Azure и SQL Server 2022.
- ◆ Developer — предоставляет всю функциональность, соответствующую версии Enterprise. При этом система лицензирована лишь для разработки и тестирования создаваемых систем.
- ◆ Express — бесплатная система, предназначенная для обучения и создания небольших приложений.

При инсталляции системы автоматически устанавливается и Azure Data Studio.

Мы будем использовать версию Developer. Требования к аппаратным и программным средствам:

- ◆ Объем внешней памяти 6 Гб, 1 Гб оперативной памяти, 64-разрядный процессор.
- ◆ Операционная система: Windows 10, Windows Server 2016.

Для работы с системой нужно на своем компьютере установить MS SQL Server 2022 Developer и Server Management Studio.

1.1. Инсталляция MS SQL Server 2022 Developer

Если у вас на компьютере были установлены предыдущие версии SQL Server, то они не мешают новой инсталляции.

В процессе инсталляции компьютер должен быть подключен к Интернету.

Нужно в браузере перейти по ссылке

<https://www.microsoft.com/en-us/sql-server/sql-server-downloads>.

В появившемся окне найти Developer (рис. 1.1) и щелкнуть по кнопке **Download now**.



Рис. 1.1. Выбор загрузки файла инсталляции

Начнется скачивание программы для установки на компьютере SQL Server. После завершения скачивания запустите на выполнение программу SQL2022-SSEI-Dev.exe (рис. 1.2).

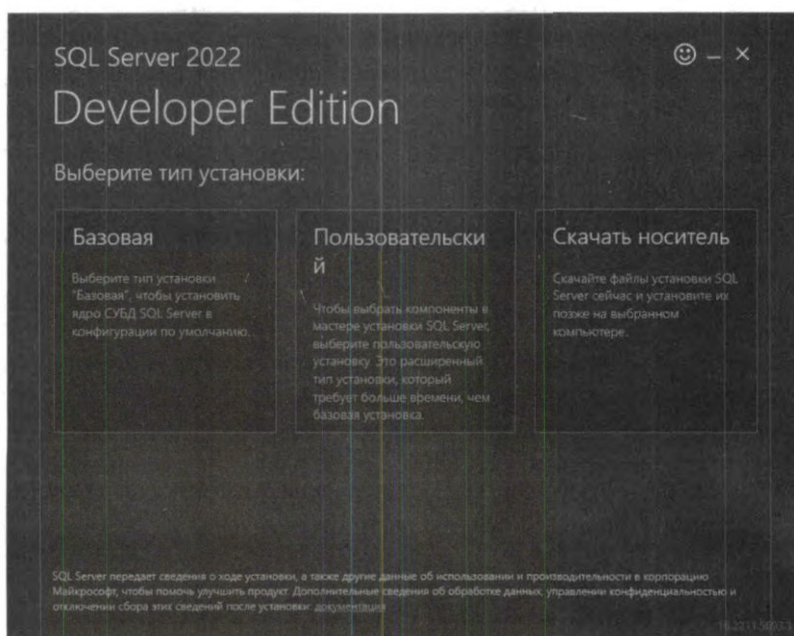


Рис. 1.2. Выбор базовой версии инсталляции

Здесь следует выбрать базовый вариант, щелкнув мышью по соответствующему изображению.

В следующем окне (рис. 1.3) нужно выбрать язык и принять лицензионное соглашение, щелкнув по кнопке **Принять**.

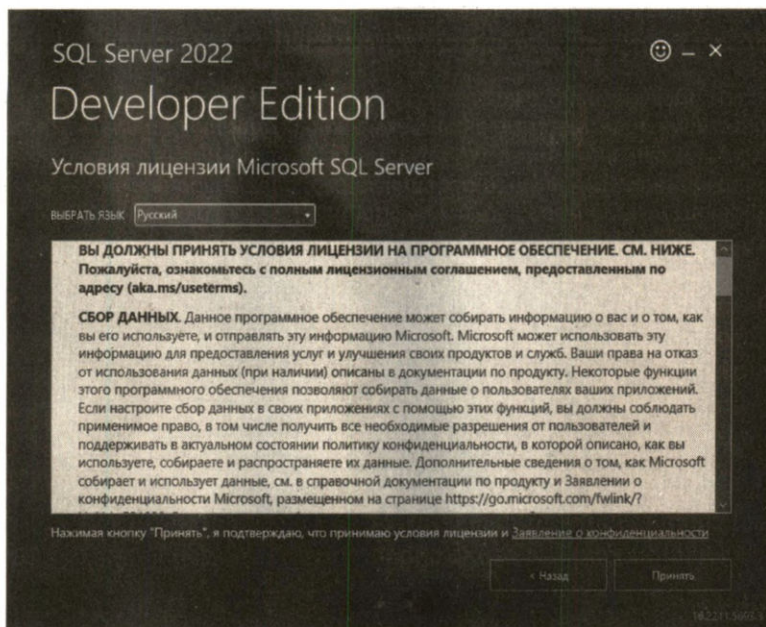


Рис. 1.3. Выбор языка, принятие лицензии

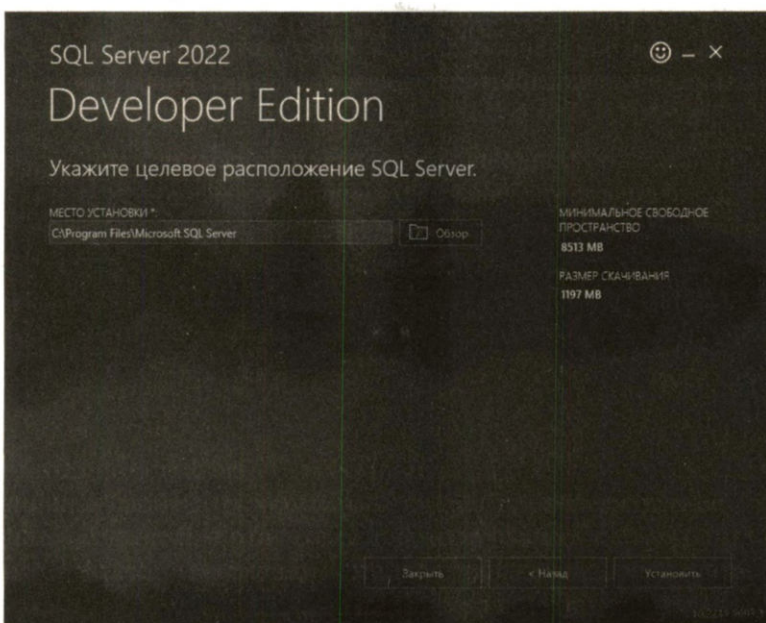


Рис. 1.4. Выбор места установки программной системы

В следующем окне (рис. 1.4) можно выбрать диск и каталог для размещения устанавливаемых программ при щелчке по кнопке **Обзор**.

После щелчка по кнопке **Установить** начинается скачивание пакета установки (рис. 1.5).

По завершении скачивания выполняется установка программного продукта (рис. 1.6).

После завершения установки появляется следующая форма (рис. 1.7).



Рис. 1.5. Процесс скачивания пакета установки



Рис. 1.6. Установка программного продукта

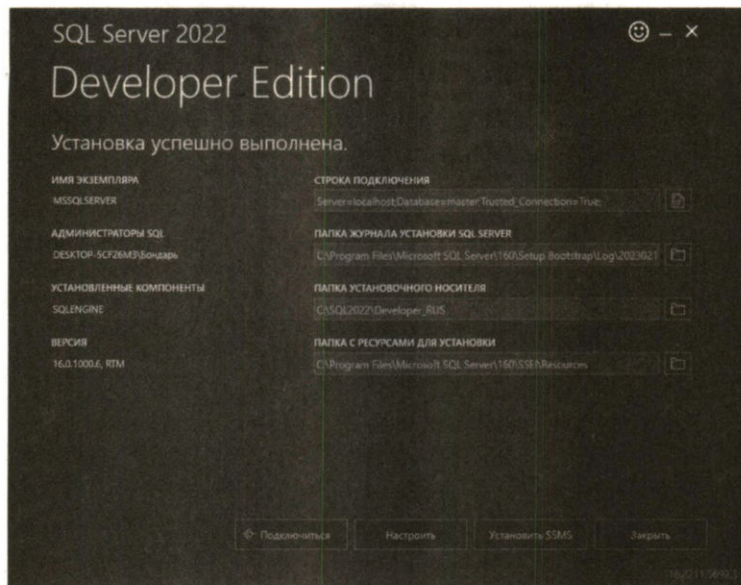


Рис. 1.7. Завершение установки

Здесь показаны имя экземпляра и строка подключения, которая будет использована при подключении к установленному серверу базы данных.

Для нормальной работы с базой данных нужно установить SQL Server Management Studio (SSMS).

1.2. Установка SQL Server Management Studio

В окне завершения установки щелкните по кнопке **Установить SSMS**.

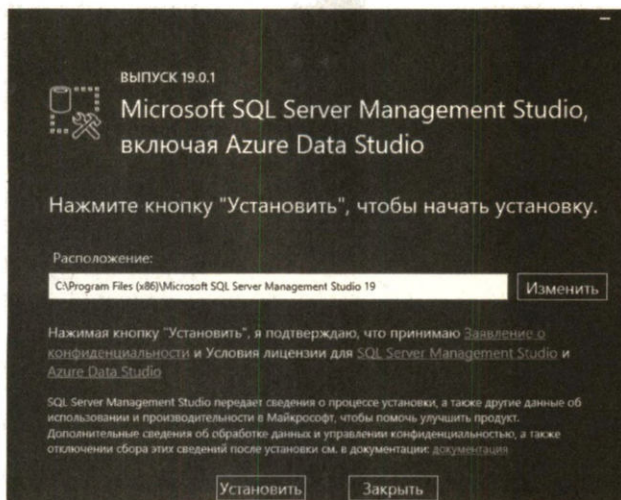


Рис. 1.8. Запрос на установку Management Studio

Будет скачан файл установки SSMS-Setup-RUS.exe.

После запуска его на выполнение появится окно запроса на установку (рис. 1.8).

Здесь можно изменить расположение устанавливаемых файлов (кнопка **Изменить**).
Далее щелчок по кнопке **Установить**.

Появляется окно загрузки пакетов (рис. 1.9).

После завершения загрузки пакетов появится окно установки Management Studio (рис. 1.10).



Рис. 1.9. Загрузка пакетов Management Studio

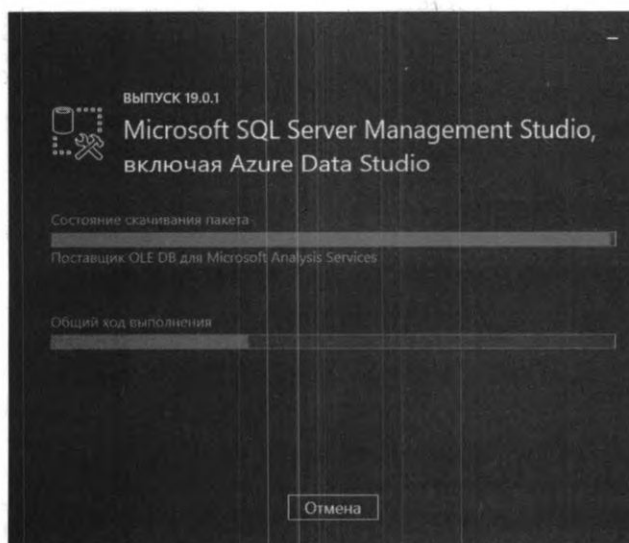


Рис. 1.10. Установка Management Studio

Затем появится окно завершения установки (рис. 1.11), в котором нужно щелкнуть по кнопке **Заккрыть**.

Для проверки правильности выполненных установок следует запустить на выполнение Management Studio (рис. 1.12).

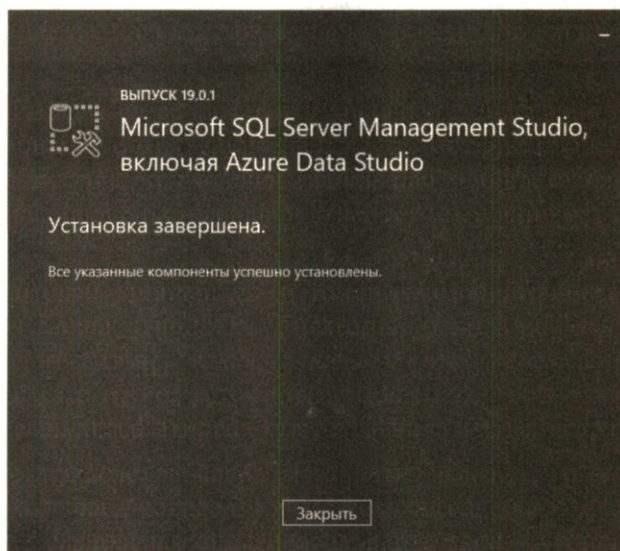


Рис. 1.11. Завершение установки Management Studio

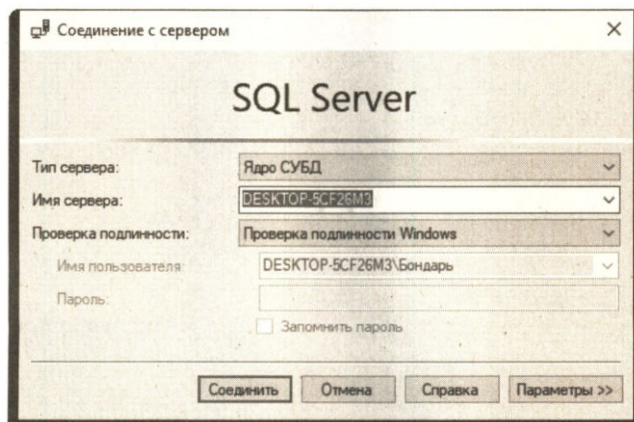


Рис. 1.12. Запуск на выполнение Management Studio

Чтобы соединиться с сервером, нужно щелкнуть по кнопке **Соединить**.

Общие сведения о SQL Server 2022

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

В этой главе мы рассмотрим основы реляционных баз данных, их объекты. Главным объектом является таблица. 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. В документации и в большей части программистской литературы для описания синтаксиса формальных языков применяются нотации Бэкуса–Наура.

Если вы знакомы с основами реляционных БД, с объектами базы данных в SQL Server, нормальными формами и процессом нормализации таблиц, то можете пропустить первые разделы этой главы или бегло их просмотреть.

2.1. Реляционные базы данных

MS SQL Server 2022 является *реляционной системой управления базами данных*, сокращенно РСУБД. В реляционных БД все данные хранятся в плоских двумерных таблицах. В основе реляционных систем лежит строгий математический аппарат.

В базе данных помимо самих данных хранятся и их описания — *метаданные*. Это позволяет уменьшить зависимость программ от данных на логическом уровне.

Основной объект реляционных БД — это таблицы.

2.1.1. Таблицы

2.1.1.1. Основные свойства и характеристики таблиц

Таблица (table) содержит произвольное число *строк* (row) или, иными словами, *записей* (record). Фактически число строк в таблице ограничивается объемом внеш-

ней 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.

Для помещения данных в таблицу служит оператор INSERT.

Данные из таблицы (таблиц) извлекаются 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 конкретной задачи предметной области. Есть умное слово, определяющее необходимые пользователю данные — *релевантные* данные. При помощи оператора SELECT мы можем выбирать данные не только из одной таблицы, но и из нескольких таблиц, используя различные средства, в первую очередь, очень удобную операцию соединения (join). Результат выборки называется *набором данных* (dataset).

Типы данных

Основная характеристика столбца — его *тип данных* (datatype). Тип данных в SQL имеет имя. Типы данных могут быть предварительно определенными в системе (predefined), их иногда называют *системными*, *встроенными* 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 появляются новые типы данных, например *XML*, или *пространственные* (spatial) типы данных. Все эти типы данных поддерживаются в 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* множество допустимых значений — это множество целых чисел в определенном диапазоне. Диапазон представления этих чисел определяется числом байтов, отводимых под целое число. Множеством допустимых операций для числовых типов данных, как целочисленных, так и дробных, являются четыре арифметические операции — сложение, вычитание, умножение и деление, а также операция получения остатка от деления.

Для числового типа данных в SQL существует очень много математических функций — возведение в различные степени, извлечение различных корней, логарифмические функции и др.

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 предусмотрено очень много полезных функций. Это выделение подстроки, удаление начальных и конечных пробелов, поиск значения в строке, замена значения подстроки и др. Все эти функции 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 БД используется еще одно значение NULL (неизвестное, неопределенное значение). Здесь применяется уже не обычная двузначная, а трехзначная логика.

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

ЗАМЕЧАНИЕ

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

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

Порядок сортировки

Порядок сортировки (collation) применяется 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 для всех баз данных, хранящихся на сервере, для конкретной БД и для отдельных строковых столбцов таблиц.

ЗАМЕЧАНИЕ

В некоторых СУБД есть 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 часто имеет несколько порядков сортировки.

Неизвестное значение NULL

Среди значений, которые может принимать столбец любого типа данных, в реляционных БД также используется и *неопределенное* 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 неизвестны 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 неизвестна. При этом большинство задач обработки данных может решаться и при отсутствии таких данных. В таком 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 данных в таблице предусмотрен столбец для указания отчества человека. У людей некоторых национальностей в принципе не бывает отчеств. 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 наличии 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 неизвестен, 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), ни FALSE (ложь), результатом окажется значение NULL. Здесь и появляется необходимость в трехзначной логике. Таблицы истинности для операций отрицания, дизъюнкции и конъюнкции в трехзначной логике будут рассмотрены в *главе 4*.

В языке SQL существуют оператор IS NULL и функция ISNULL(), выполняющие проверку на неизвестное значение. Поскольку в языках программирования принята (пока еще) двузначная логика, то при работе с базами данных при сравнении значений столбцов до применения обычных операций сравнения регулярно вызывается функция ISNULL() или оператор IS NULL.

ЗАМЕЧАНИЕ

Вы можете встретить в литературе критические замечания относительно использования в базах данных и вообще в программировании значения NULL. Я считаю, что не следует принимать близко к сердцу негодующие высказывания по этому поводу, даже если они исходят от известных специалистов в области программирования и баз данных. Практический опыт свидетельствует, что в разработках существует обоснованная потребность в значении NULL.

Индексы

Объект базы данных *индекс* (index) предназначен для отдельных таблиц. Для каждой таблицы в MS SQL Server 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 процессы выборки данных из таблицы и упорядочивания выбранных данных, а также дают возможность обеспечить уникальность 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. Таблица, не имеющая кластерного индекса, называется *кучей* (heap).

Индексы создаются разработчиками базы данных для отдельных таблиц. В некоторых случаях система автоматически создает индексы для ключей таблицы, в первую очередь, для первичных ключей.

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. Основное требование к первичному ключу — его *уникальность*. Иными словами, в таблице не должно быть двух различных строк с одинаковым значением первичного ключа. Ни один столбец, входящий в состав первичного ключа, не может иметь значения NULL (в описании таких столбцов должно присутствовать предложение NOT NULL).

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

СУБД 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, что показала практика использования реляционных БД. Забегая 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 число *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. Одно из назначений уникальных ключей — устранение дублирования значений, как и в случае уникальных индексов.

СУБД 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 той 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.

ЗАМЕЧАНИЕ

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 всё 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 со значением в точности равным значению этого внешнего ключа.

Отношения между таблицами в базе данных

Важнейшими отношениями (связями) в реляционных БД 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 БД в случае правильного ее проектирования.

Декларативная целостность базы данных обеспечивается системой управления базами данных. СУБД отменяет все попытки добавления и изменения данных, которые нарушают заданную средствами операторов DDL (Data Definition Language — язык определения данных) целостность (т. е. непротиворечивость) данных — в БД 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. 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 условие указанного ограничения. 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, 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 БД, 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. Представление позволяет скрыть от пользователя сложный процесс выборки данных. Кроме того, оно 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 пред-

ставлению, как и в случае обычной выборки данных из таблицы с помощью оператора `SELECT`, является набор данных.

Представления бывают *изменяемые* и *неизменяемые*. Изменяемое представление позволяет программе, вызвавшей представление, 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 БД и 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, циклы и другие императивные средства. Язык допускает рекурсию, т. е. тот случай, когда программа вызывает саму себя.

Этот язык применяется 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. Это очень полезные средства, особенно при проверке и отладке операторов выборки данных, да и вообще для работы с данными в БД. Такая возможность существует не во всех СУБД.

Хранимые процедуры (`stored procedure`) представляют собой программы, хранящиеся в БД и выполняющие различные действия, обычно с данными из базы данных, хотя процедуры могут и не осуществлять никаких обращений к БД. К хранимым процедурам могут обращаться любые программы, работающие с базой данных, к ним также могут обращаться и другие хранимые процедуры и триггеры. Допустима 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 большой по объему БД 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 заданные 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 ("вспыхивают" — `fire`) при наступлении некоторого события в БД — при добавлении, изменении или удалении строк конкретной 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 в БД, которые 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 БД: 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 большого количества 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 авторизацию пользователей рекомендуется выполнять средствами 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 БД, 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 БД, которому назначаются некоторые полномочия к отдельным объектам базы данных. Затем роль может назначаться различным пользователям. В момент соединения с БД при указании роли пользователь получает все полномочия, предоставленные данной роли.

2.1.5. Задание первичных ключей таблиц

В каждой таблице должен быть 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 с одинаковыми значениями 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 индек-

сы. 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.

Пусть, например, для таблицы, описывающей людей (персонал организации, студенты в учебном заведении), нужно выбрать столбцы, которые войдут в состав первичного ключа. Понятно, что использовать в этом качестве фамилию нельзя: существует слишком много 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 (например, у меня был студент Михаил Сергеевич Горбачев). Кроме того, размер первичного ключа получается слишком большим (не менее 50 символов), что отрицательно скажется на объеме требуемой внешней памяти и на производительности системы.

Для сотрудников организации с этой целью подойдет табельный номер, если он уникален 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 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 *искусственный первичный ключ*. Иногда в литературе можно встретить термин "суррогатный" (surrogate) первичный ключ, который не очень нравится русскоязычным программистам из-за 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 искусственного первичного ключа — формирование последовательностей (sequence) для получения уникального значения.

Что касается всевозможных вспомогательных, связующих, таблиц, которые вскоре мы рассмотрим, то для них довольно часто 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 БД. Все операторы работы с базой данных (как с данными, так и с метаданными) выполняются

в рамках (или, как 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 БД 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 видны другим процессам, но об этом потом). Отмененные действия не сохраняются в БД.

Транзакциям можно задать некоторые характеристики, которые определяют поведение транзакции по отношению к другим параллельным процессам, а также допустимые одновременные действия других процессов.

Транзакции — важное средство обеспечения одновременной работы с БД большого числа 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 попросту 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 реализует 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 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.

Существует также механизм "грязного чтения". В этом случае другие пользователи могут видеть неподтвержденные изменения записей. Об уровнях изоляции транзакций более подробно поговорим в *главе 10*.

2.1.7. 12 правил Кодда

Очень часто при описании реляционных систем управления базами данных приводят известные 12 правил Кодда, которые нужны в первую очередь разработчикам самих СУБД. Но нам с вами, специалистам по работе с уже созданными 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 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 БД 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 описывать данные графически. Прямоугольники изображают таблицы, линии со стрелками или без стрелок описывают 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. Основные причины использования этого отношения — экономия памяти и увеличение скорости выполнения запросов. Такое отношение целесообразно в случае, если связь между двумя таблицами не обязательна. В реальной жизни подобные случаи встречаются не так часто.

Рассмотрим связь между человеком и его паспортными данными. Это связь "один к одному". В данном случае необходимы две таблицы. Одна содержит какие-то сведения о человеке, другая — сведения о его паспортных данных (номер паспорта, дата выдачи, кем выдан и др.). Такая связь между двумя таблицами представляется на диаграмме линией без стрелок (рис. 2.1).



Рис. 2.1. Пример отношения один к одному

Если паспортные данные поместить в первую таблицу, то при отсутствии соответствующих сведений это привело бы к появлению пустых полей в таблице и перерасходу внешней памяти. Нужно сказать, что на самом деле на это не потребуется большого объема памяти, поскольку физическое хранение в SQL Server хорошо продумано, и пустые поля занимают минимальное место во внешней памяти.

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

Можно привести еще некоторые примеры 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". Иногда в литературе это отношение 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 показано на *рис. 2.2*.



Рис. 2.2. Пример отношения "один ко многим"

2.2.3. Отношение "многие ко многим"

Хороший пример такого отношения — отношение между таблицей авторов и таблицей книг. Одна книга может быть написана несколькими авторами, один автор может написать несколько книг (*рис. 2.3*).

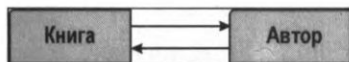


Рис. 2.3. Пример отношения "многие ко многим"

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

Отношение "многие ко многим" реализуется добавлением в БД третьей, связующей таблицы и установлением двух необходимых связей "один ко многим".

В рассматриваемом 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).

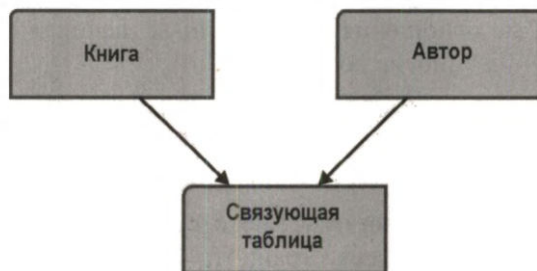


Рис. 2.4. Связующая таблица для реализации отношения "многие ко многим"

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

Столбец "Код книги" в связующей таблице — это внешний ключ, 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 уникальна и может служить первичным ключом, однако часто существуют и иные ситуации, где для связующей таблицы приходится вводить свой искусственный первичный ключ.

Нужно упомянуть, что в библиографических информационно-поисковых системах такие отношения ("многие ко многим"), реализуемые при помощи добавления связующих таблиц, встречаются постоянно. Например, отношения "книга – переводчик", "книга – научный редактор", "книга – ключевое слово". А вот "книга – издательство" — это отношение "многие к одному".

2.3. Нормализация таблиц

2.3.1. Цель нормализации таблиц

Первая задача при проектировании БД на логическом уровне — составление списка таблиц и разработка структуры каждой таблицы (об уровнях проектирования см. разд. "2.4. Проектирование баз данных"). В реляционных БД есть понятие нормализации таблиц. Существует набор стандартов проектирования данных, называемый нормальными формами. Нормальные формы определяют правила, которым должны соответствовать структуры таблиц. Общеизвестны шесть нормальных

форм, хотя в литературе по базам данных можно найти и гораздо больше форм нормализации. На практике обычно применяется 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 ее на две или более таблиц с меньшим числом столбцов. Эти таблицы для наглядности отображения данных пользователю за счет связи "внешний ключ / первичный ключ" или "внешний ключ / уникальный ключ" снова можно соединить в процессе выборки данных в операторе `SELECT` при помощи операции соединения (`JOIN`).

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

Опыт проектирования БД показывает, что правильное выполнение нормализации таблиц реально создает удобную в эксплуатации базу данных и позволяет осуществлять к нормализованной базе данных даже такие 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 БД были созданы по правилам нормализации.

ЗАМЕЧАНИЕ

Я часто вспоминаю одну историю. Для некоторой организации была создана программная система для решения довольно сложных задач. Руководительница организации была в восторге от реализованной функциональности. Затем, помявшись, она спрашивает, а можно ли добавить еще одну нужную выборку данных, что не было предусмотрено договором. Я ей сказал, что завтра привезем ей расширенную систему. Она решила, что это шутка. Поскольку все таблицы были созданы в третьей нормальной форме, добавление новой выборки данных не потребовало много времени. На следующее утро продемонстрировали ей эту функцию. Она была в еще большем восторге.

2.3.2. Первая нормальная форма

Первая нормальная форма (иногда в литературе встречается 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 дополнительное разумное требование, чтобы каждая таблица имела первичный ключ.

В стандарте SQL2008 атомарность определяется как значение, которое нельзя разделить 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 можно разделить 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.

Рассмотрим пример. Пусть в базе данных имеется справочник стран — таблица COUNTRY. На начальном этапе проектирования некоторой системы обработки данных было ясно, что для решения задач предметной области нужна не столько сама страна, сколько список всех ее регионов. Поэтому 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
		...
		Симферополь
		...

Первичным ключом здесь является столбец "Код страны".

Таблица включает 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 первичного ключа этой новой таблицы нужно помимо кода страны задать еще и код региона. Несколько строк такой таблицы показано в табл. 2.2.

Таблица 2.2. Таблица регионов

Код страны	Код региона	Центр региона
RUS	32	Брянск
RUS	25	Владивосток
RUS	15	Владикавказ

Таблица 2.2 (окончание)

Код страны	Код региона	Центр региона
RUS	33	Владимир
RUS	34	Волгоград
RUS	82	Симферополь

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.

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 случаю, когда первичный ключ образован из нескольких столбцов.

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
RUS	82	Симферополь	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 дополнительного ограничения: ни один неключевой столбец не дол-

жен зависеть 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 — это первичный ключ. 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. Множество других "нормальных" форм, скорее всего, существуют лишь в теории. 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 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, помимо первичного ключа. При этом общий объем данных в БД за счет большого числа таблиц сильно увеличивается, что, как правило, ухудшает производительность системы.

В реальной жизни пятая форма практически не встречается.

2.3.6. Денормализация таблиц

Нормализованные таблицы в базе данных позволяют уменьшить избыточность данных, в большинстве случаев увеличивают гибкость взаимодействия с системой, 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 станет не создание отдельной таблицы, где будет храниться температура конкретного пункта в конкретное время, а наличие в основной таблице в качестве столбца массива, содержащего 24 элемента, по одному 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 многочисленных операций с очень большим объемом данных в БД, часто отходят от требований нормализации. При проектировании подобных БД 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, но 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 не сама по себе БД, "вещь в себе", а с учетом тех задач, для решения которых она будет использоваться.

Один из наиболее распространенных подходов — трехуровневое проектирование всей системы обработки данных и, соответственно, базы данных. Это концептуальный (содержательный), логический и физический уровни.

На первом, концептуальном, уровне анализируют предметную область, для решения задач которой проектируется система обработки данных и БД. Выявляют и описывают объекты (object), или сущности (entity) предметной области, их свойст-

ва, атрибуты (attribute), устанавливают связи, отношения (relationship) между сущностями, определяют список задач обработки данных, фиксируют 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 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 предназначены для хранения сведений об этих объектах.

На втором, логическом, уровне создают *логическую модель* базы данных. Здесь происходит детализация концептуальной модели на основе принципов и понятий реляционных БД.

Каждый объект предметной области представляют в виде одной или более взаимосвязанных таблиц. Для каждой таблицы описывают ее структуру, где помимо основных характеристик (столбцов таблицы) определяют ключевые реквизиты. Это первичные и внешние ключи, которые позволяют однозначно идентифицировать конкретный элемент объекта (строку таблицы) и задать связи (отношения) между таблицами.

В результате появляется *логическая модель* данных, которая описывается в терминах реляционной СУБД.

На *физическом уровне* окончательно формируют характеристики объектов базы данных. Здесь задают физические характеристики всех объектов: количество и физические характеристики файлов БД, уточняют типы данных и размерность полей в таблицах. При необходимости создают дополнительные объекты базы данных — генераторы, триггеры, хранимые процедуры, представления, для таблиц создают индексы, позволяющие повысить производительность системы обработки данных.

MS SQL Server предоставляет средства тонкой настройки физических характеристик базы данных, позволяя для одной БД 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 и отказоустойчивость системы.

На практике при создании отдельных систем обработки данных то ли по причине дефицита времени, то ли от врожденной лени многие разработчики часто опускают формальную фиксацию результатов концептуального проектирования, ограничиваясь лишь общим описанием содержательных требований к системе, к обрабатываемым данным. Нужно сказать, что в большинстве случаев это оправданно, поскольку разработчики оперируют в своей деятельности понятиями, в первую очередь, логического уровня (таблицы, столбцы, ограничения и т. д.) и результаты их проектирования удовлетворяют всем требованиям к системе. Правда, своих сту-

дентов я заставляю выполнять концептуальное проектирование данных и описывать его результаты. И эти результаты впечатляют.

ЗАМЕЧАНИЕ

В существующей литературе иногда можно встретить уровни проектирования, называемые инфологическим и даталогическим. Не рекомендую использовать эту терминологию и то, что там предлагается делать.

2.5. Язык Transact-SQL

В SQL Server для работы с базами данных используется язык, который называется *Transact-SQL*. Этот язык представляет собой несколько измененный и расширенный вариант языка SQL, определенный в международных стандартах. Еще говорят, что Transact-SQL является *диалектом* стандарта SQL. Операторы языка позволяют создавать, изменять и удалять объекты БД и саму базу данных, создавать триггеры, хранимые процедуры, добавлять, изменять, удалять и отыскивать данные в таблицах БД.

Основная законченная единица языка Transact-SQL — оператор (statement), который может состоять из нескольких предложений (clause). При написании операторов и предложений используются константы (литералы) и ключевые слова. Это слова, которые недопустимы в качестве имен объектов базы данных. (Вообще-то это не совсем так. Ключевые слова возможны 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 во много 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 существуют довольно простые и удобные средства, которые мы далее рассмотрим.

Семантика — это смысл синтаксически правильно построенных языковых конструкций. Семантика конструкций описывается, объясняется с помощью обычного естественного языка. При описании семантики, например, оператора `CREATE DATABASE` можно сказать, что он позволяет создать базу данных для текущего экземпляра сервера БД (или подключить базу данных к списку доступных БД), 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 воспользуемся несколько расширенной системой обозначений (нотации) Бэкуса–Наура,

что принято во всем мире при описании синтаксиса большинства формальных языков программирования.

ЗАМЕЧАНИЕ

В предыдущих своих книгах я также использовал графические средства — R-графы. Это довольно наглядный способ описывать синтаксические конструкции. В этой книге от данного средства я отказываюсь, в первую очередь, чтобы сократить объем передаваемой вам информации.

В нотациях Бэкуса–Наура в угловые скобки < и > заключают определяемый синтаксический элемент, конструкцию или понятие языка. Например, конструкция <идентификатор> задает понятие идентификатора в Transact-SQL. Такая конструкция называется "нетерминальным символом", т. е. символом, выражением, определяемым в дальнейшем 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" — 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 в дальнейшем нужно определить 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 список элементов, разделенных символом вертикальной черты |, заключен в фигурные скобки { и }, то из этого списка должен быть выбран в точности один элемент. Если же такой список заключен в квадратные скобки [и], 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 подчеркнут, то это элемент 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 (MB), так как в синтаксической конструкции это значение подчеркнуто. Запись

SIZE = 18 MB

эквивалентна записи

SIZE = 18

Хочу сказать, что ориентация при написании операторов любого языка на некоторые значения по умолчанию не такое уж хорошее дело. Бывали случаи, когда в новых версиях систем значения по умолчанию изменялись, что 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. Лучшее решение все-таки — 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 число раз.

Пример — следующая 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 число параметров, отделенных друг от друга запятыми. Круглые скобки нужны в любом случае.

2.5.1.1. Числа

Целое число

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

`<целое> ::= <цифра>...`

Это цифра (десятичная цифра), которая может повторяться произвольное число раз, но не менее одного раза.

Целое со знаком

Целое со знаком — это целое число, перед которым стоит знак числа: + или -:

`<целое со знаком> ::= {+ | -}<целое>`

Дробное число

Дробное число может содержать знак числа, целую часть (необязательно) и после десятичной точки — дробную часть:

`{+ | -}[<целое1>].<целое2>`

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>}{Е | е}{+ | -}<целое3>`

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

Числу с плавающей точкой может предшествовать знак числа. Знак может отсутствовать (знаки заключены в квадратные скобки). Само число может содержать целую и дробную части, разделенные десятичной точкой, причем любая часть может отсутствовать. Целая и дробная части могут отсутствовать. После этого идет 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 0х, за которыми следует не менее одной шестнадцатеричной цифры:

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

ЗАМЕЧАНИЕ

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

2.5.1.2. Комментарии

В SQL существует возможность в любой позиции, где допустимы пробелы, задавать комментарии. Произвольный текст можно заключить между символами `/*` и `*/`. Этот текст служит только для объяснения, что конкретно содержится или выполняется в соответствующем операторе. Такой комментарий может занимать произвольное число 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.

2.5.1.3. Строковые константы

Строковые константы можно всегда заключать в апострофы. Строка 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 2014'  
'Ann's daughter'  
SET QUOTED_IDENTIFIER OFF;  
GO  
"Ann's daughter"
```

Настоятельно рекомендую всегда заключать строковые константы только в апострофы. В некоторых СУБД 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 точка 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 точка 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.

ЗАМЕЧАНИЕ

В MS SQL Server любой версии оператор не обязательно должен завершаться символом точка с запятой. Его отсутствие не вызовет ошибки. Однако лучше вести себя прилично и не нарушать требования стандарта.

2.5.1.4. Идентификатор

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

Все объекты базы данных (сама БД, ее логические файлы, 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) допустимо явно не указывать имена — 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 без пробелов, специальных символов, в том числе некоторых разделителей. Они не могут быть зарезервированными словами языка Transact-SQL.

В различных языках программирования и в разных СУБД правила задания идентификаторов несколько отличаются.

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

Правила записи *обычных идентификаторов* похожи 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 допустимы буквы 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 одинаковые. Это также верно и для идентификаторов, в которых присутствуют буквы кириллицы. Например, идентификаторы Таблица1 и таблица1 будут рассматриваться системой как одинаковые.

Я рекомендую считать, что все идентификаторы *чувствительны* к регистру. Если объявили какой-нибудь объект, то советую при обращении к нему использовать

именно тот вариант написания имени, который был задан при объявлении объекта. Часто, если вы переносите скрипт из одной системы в другую, в той системе имена могут быть чувствительны к регистру.

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

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

Обычный идентификатор не может быть зарезервированным словом языка Transact-SQL. Список зарезервированных слов приведен 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 (l) и правой (r) квадратными скобками. Есть два случая, когда необходимы идентификаторы с разделителями.

- ◆ В качестве идентификатора зарезервированных слов (ну никак не могу одобрить такое их использование, хотя наблюдал ситуации, когда необходимость применения этих слов довольно правдоподобно объяснялась разработчиками конкретных баз данных).
- ◆ Для именования объектов базы данных, содержащих пробелы или символы, отличные от допустимых в обычных именах. При таком способе именования объектов можно создавать имена, понятные любому непосвященному во все премудрости программирования человеку.

Идентификаторы с разделителями *чувствительны* к регистру. Все приведенные далее идентификаторы "name1", "NAME1", [name2], [NAME2] являются различными. 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 допустимы только в том случае, когда значение опции QUOTED_IDENTIFIER для текущего соединения с сервером БД установлено в ON (значение по умолчанию). При этом строковые константы можно заключать только в апострофы, но не в кавычки. Если же значение опции QUOTED_IDENTIFIER задано как OFF, то кавычки для задания идентификатора с разделителями не допускаются. В этом случае для идентификаторов с разделителями возможны только квадратные скобки, а для задания строковых литералов допустимы как апострофы, так и кавычки. Нужно заметить, что эти правила 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, требуется задать символ правой (только правой) квадратной скобки `]`, то этот символ нужно записать дважды. Например, для задания имени

Код `[Code]` вида деятельности

следует 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 предпочтительнее 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>$
```

Из приведенного фрагмента 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.

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

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

Теперь все в порядке.

Все идентификаторы (как обычные, так и с разделителями) для "нормальных", наиболее распространенных объектов базы данных могут содержать не более 128 символов. Число символов в идентификаторах, предназначенных для именования временных объектов, не может превышать 116. При подсчете числа 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 число символов, отводимых для именования объектов БД. Насколько могу вспомнить, речь шла об увеличении числа символов со 128 до 256. Лично я сильно сомневаюсь, что большое число символов в именованиях объектов поможет улучшить понимание состава и назначения объектов БД. В моей практике не встречалось объектов, размер имени которых превышал бы 30 символов.

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 восприятия 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 употребляются 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 подязыков, частей. По традиции каждый такой подязык 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 предусмотрены следующие группы операторов:

- ◆ CREATE — это операторы, при помощи которых создаются новые объекты БД (в первую очередь таблицы, затем пользовательские типы данных, индексы, хранимые процедуры, триггеры, роли и др.). С помощью оператора CREATE DATABASE создается и сама база данных.
- ◆ DROP — операторы этого вида удаляют ранее созданные ненужные, как потом выяснилось, объекты БД: таблицы, пользовательские типы данных и иные объекты. Оператор позволяет удалить и базу данных.
- ◆ ALTER — эти операторы позволяют изменить уже существующие в базе данных ранее созданные объекты и характеристики БД.

Для работы с собственно данными в БД существуют операторы DML, позволяющие создавать, изменять и удалять данные. В состав DML входит оператор, выполняющий одну из наиболее важных функций в базе данных — поиск (выборку) данных.

Для *данных* в БД применяются следующие четыре основных оператора:

- ◆ INSERT — добавление данных;
- ◆ UPDATE — изменение существующих данных;
- ◆ DELETE — удаление данных;
- ◆ SELECT — выборка (поиск) данных.

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:

- ◆ GRANT — предоставление полномочий для доступа к защищаемому объекту;
- ◆ DENY — отмена полномочий;
- ◆ REVOKE — удаление полномочий.

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

- ◆ BEGIN TRANSACTION и BEGIN DISTRIBUTED TRANSACTION — старт обычной или распределенной транзакции;

- ◆ COMMIT TRANSACTION, COMMIT WORK — подтверждение транзакции;
- ◆ ROLLBACK TRANSACTION, ROLLBACK WORK — откат транзакции;
- ◆ SAVE TRANSACTION — создание точки сохранения.

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

Что дальше?

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

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 диалоговых средств 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 БД. В ней будут храниться все данные, необходимые для решения задач предметной области. Туда же вы можете помещать создаваемые хранимые процедуры, триггеры, пользовательские типы данных, функции и представления.

Здесь мы рассмотрим не только создание, изменение и удаление БД. Существующий в языке Transact-SQL оператор `CREATE DATABASE`, который в первую очередь должен создавать базу 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 БД (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 БД.

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. Для создания баз данных и изменения их характеристик также применим диалоговые средства системы, представленные в этом наиболее важном и удобном в работе графическом инструменте администрирования и разработки SQL Server — Management Studio. Рекомендую выполнить на вашем компьютере все 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 — довольно сложный объект с множеством характеристик. Мы все 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 БД на профессиональном уровне.

3.1. Запуск и останов экземпляра сервера

При первоначальной загрузке вашего компьютера, поскольку при установке SQL Server был задан автоматический старт сервера БД, 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 БД, с которым мы выполняем соединение в одной из наших программ (это будет утилита командной строки sqlcmd и программа SQL Server Management Studio), называется *текущим экземпляром сервера*.

Разные версии SQL Server позволяют иметь от 16 до 50 экземпляров сервера. 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.

3.1.1. Запуск экземпляра сервера

Если в процессе работы компьютера вы остановили SQL Server, то экземпляр нужно запустить вручную. Проще это сделать при помощи Management Studio.

Запустить на выполнение сервер можно при помощи компонента SQL Server Management Studio. При вызове программы появится диалоговое окно соединения с сервером (рис. 3.1).

Здесь нужно щелкнуть по кнопке **Отмена**, чтобы отменить соединение с сервером, который пока еще не запущен на выполнение. Появится главное окно программы Management Studio. В главном меню нужно выбрать элементы **Вид | Зарегистрированные серверы**.

Чтобы получить список экземпляров серверов SQL Server, установленных на компьютере, нужно в левой части окна, в панели **Зарегистрированные серверы** рас-

крыть элемент **Ядро СУБД**, дважды щелкнув по нему мышью или щелкнув по символу + слева от имени этого элемента, затем таким же способом раскрыть элемент **Группы локальных серверов** (рис. 3.2). На рисунке видно, что на компьютере зарегистрировано два экземпляра серверов, и оба выполняются.

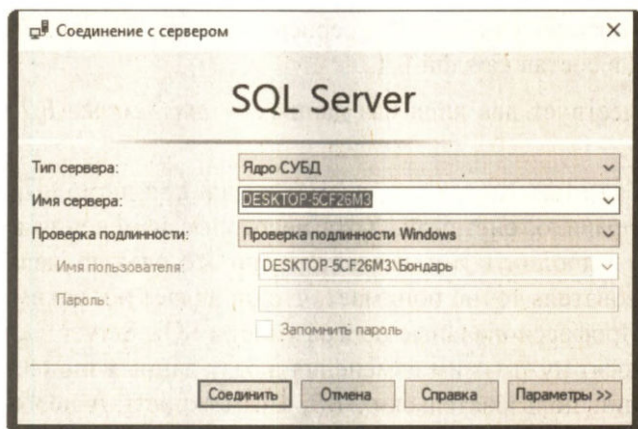


Рис. 3.1. Диалоговое окно соединения с сервером

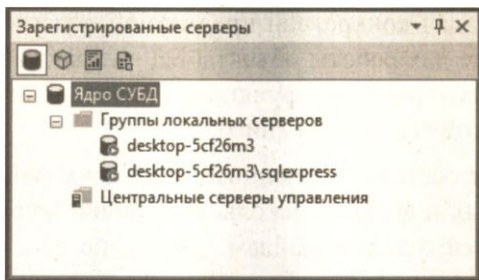


Рис. 3.2. Список зарегистрированных экземпляров серверов SQL Server

В списке зарегистрированных серверов щелкните правой кнопкой мыши по имени сервера и в контекстном меню выберите элемент **Управление службой**, в появившемся подменю выберите элемент **Запустить** для запуска сервера. Появится окно подтверждения, в котором нужно щелкнуть по кнопке **Да**.

3.1.2. Останов экземпляра сервера

Для останова сервера в левой панели главного окна Management Studio нужно щелкнуть правой кнопкой мыши по имени сервера (см. рис. 3.2), в контекстном меню выбрать элемент **Управление службой**, в появившемся подменю выбрать элемент **Останов**. Появится диалоговое окно подтверждения, 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 БД, 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 БД.

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

Системные базы данных создаются автоматически при установке SQL Server и изменяются, как правило, системой. Хотя некоторые изменения в отдельных системных БД 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 установки системы вносят нужные им изменения в базу данных *model*, которая используется при создании пользовательских БД. Возможность и необходимость таких изменений рассматривается в этой главе и в *приложении 4*.

Пользовательские базы данных создаются, изменяются и удаляются пользователем. В такие БД вы помещаете все данные, которые необходимы вам (вашему заказчику) для решения задач конкретной предметной области. Сюда же можно помещать и другие нужные для работы объекты БД — хранимые процедуры, пользовательские типы данных, триггеры, функции. Именно этими базами данных мы будем заниматься на протяжении всей книги.

Любая БД 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 БД, будь то 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 рассмотрим в этой главе, когда будем создавать и изменять БД, а также отображать сведения об этих базах данных и об их файлах. Полный, или почти полный перечень характеристик БД, а также перечень существующих в системе средств изменения и отображения этих характеристик приведен в *приложении 4*.

К базе данных можно обратиться по ее логическому имени, задаваемому при создании БД. При обращении к некоторым функциям необходим идентификатор базы данных — целочисленное значение, присваиваемое системой в момент создания БД. 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 использования мы рассмотрим чуть позже в этой главе.

3.2.1. Системные базы данных

Системные БД содержат сведения, необходимые для работы SQL Server. К таким БД относятся:

- ◆ 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 БД, 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.

3.2.1.1. База данных master

База данных master, пожалуй, наиболее важная системная БД в SQL Server. Она содержит все данные, необходимые для работы с СУБД. В ней также хранятся данные о конфигурации сервера БД, сведения обо всех пользовательских базах данных, созданных в экземпляре сервера: характеристики БД, характеристики и размещение файлов каждой БД. Настоятельно рекомендуется выполнять резервное копирование базы master при создании, изменении или удалении любой БД пользователя. SQL Server не сможет выполняться, если база данных master недоступна.

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

3.2.1.2. База данных model

Основное назначение базы данных model — хранение шаблонов для всех вновь создаваемых пользователем баз данных. При создании новой пользовательской БД в нее из базы 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 БД model, то эти объекты будут копироваться во все вновь создаваемые пользовательские БД 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).

3.2.1.3. База данных msdb

В SQL Server существуют средства создания расписаний (schedule) для автоматического исполнения заданий, ведения истории их выполнения и для выдачи предупреждающих сообщений (alert). Все это хранится в базе данных msdb и используется в основном компонентом SQL Server Agent. В этой БД также хранится история создания резервных копий, SSIS пакеты, сведения о репликациях. Эти данные предназначены для компонентов Service Broker и database mail.

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

3.2.1.4. 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 БД при выполнении запросов, а также ряд других объектов. Во многих случаях база данных tempdb позволяет повысить производительность системы при выполнении различных операций с БД.

Эта база данных так же, как и все остальные системные БД, состоит из двух файлов: файла данных (логическое имя tempdev, имя файла tempdb.mdf) и журнала транзакций (логическое имя templog, имя файла templog.ldf).

3.2.1.5. База данных resource

В SQL Server существует и такой невидимый обычными средствами объект, как скрытая база данных resource. Эта БД 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 БД хранится отдельно от всех других системных баз данных. Несмотря на то, что она сильно "засекречена", сообщу вам имена ее файлов (никому не говорите): файл данных имеет физическое имя mssqlsystemresource.mdf, файл журнала транзакций — физическое имя mssqlsystemresource.ldf.

Не следует пытаться вручную вносить изменения в системные БД. Требуется только 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 БД 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, на-

пример создаю БД со всеми значениями по умолчанию, заданными в базе данных *model*, а уже потом вношу нужные мне изменения. Более того, если хочется сразу при создании новой БД задать множество нестандартных значений для ее характеристик, то для этого можно задействовать диалоговые средства *Management Studio*.

Дальше вам решать. *It's up to you.*

3.2.2. Базы данных пользователей

В созданной пользователем базе данных хранится множество объектов. Главный объект, разумеется, таблицы, в которые вы помещаете все данные, необходимые для решения задач конкретной предметной области. Кроме таблиц БД 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 — важнейший принцип, применяемый во всех БД. Это позволяет уменьшить зависимость программ от структуры базы данных.

Помимо пользовательских объектов в базе данных хранятся системные объекты — системные таблицы, системные представления (их очень много), 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 и схемы.

Для любой БД требуется 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", БД 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 Мб.

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 БД 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, чтобы повысить производительность 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* составляет 8 Кб и его нельзя изменить.

Журнал транзакций также может быть представлен несколькими файлами. В журнале хранятся все изменения базы данных, выполненные в контексте каждой тран-

зации. Прежде чем записать выполненные пользователем изменения в файл данных, система вначале осуществляет необходимые записи в журнал транзакций. Журнал служит для выполнения операций подтверждения (COMMIT) или отката (ROLLBACK) транзакций, а также для восстановления БД в 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 Кб.

Для одного экземпляра сервера БД 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 число баз данных в одном экземпляре сервера и такое число файлов в базе данных.

3.2.3. Некоторые характеристики базы данных

Каждая база данных имеет множество характеристик. Характеристики БД, 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 БД (owner) имеет все полномочия к базе данных. Он может изменять характеристики базы, удалять ее, вносить любые изменения в данные и метаданные. Владелец БД становится пользователем, 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 БД 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 данных 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 БД. 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 быть доступной 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 БД находится в одном конкретном состоянии (state). В SQL Server существуют следующие состояния базы данных:

- ◆ **ONLINE** — база данных находится в доступном состоянии (в другой терминологии — 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 БД).
- ◆ **OFFLINE** — база данных в недоступном состоянии (или еще говорят, что она находится в автономном режиме). Никакие действия с объектами БД 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 БД автоматически будет переведена в оперативное состояние (**ONLINE**).
- ◆ **RECOVERY_PENDING** — это состояние ожидания исправления ошибок восстановления базы данных. База данных недоступна. В процессе восстановления БД 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 поврежденной. Со стороны пользователя требуются действия по устранению ошибок.
- ◆ **EMERGENCY** — база данных повреждена и находится в состоянии только для чтения (`READ_ONLY`). Такое состояние БД 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 БД.

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 БД 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). Можно задать конкретную величину, ограничив максимальный объем памяти для хранения файла, или указать, что размер файла не ограничивается (unlimited).

3.2.4.2. Состояния файлов базы данных

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

Возможные состояния файла базы данных:

- ◆ **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 недоступны, то все же возможны различные операции с БД, 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 помощи множества разнообразных средств, входящих в состав SQL Server:

- ◆ 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 к системному представлению просмотра каталогов *sys.databases*:

```
SELECT * FROM sys.databases;
```

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

- ◆ *name* — содержит логическое имя БД.
- ◆ *database_id* — идентификатор БД (целое число), автоматически присваиваемый базе данных системой при ее создании.
- ◆ *create_date* — дата и время создания БД. Время указывается с точностью до миллисекунд.

- ◆ `collation_name` — имя порядка сортировки по умолчанию для БД.
- ◆ `is_read_only` — определяет, является ли БД базой только для чтения:
 - 0 — БД находится в режиме только для чтения (`READ_ONLY`);
 - 1 — БД в режиме чтения и записи (`READ_WRITE`).
- ◆ `state` — состояние БД:
 - 0 — `ONLINE`;
 - 1 — `RESTORING`;
 - 2 — `RECOVERING`;
 - 3 — `RECOVERY_PENDING`;
 - 4 — `SUSPECT`;
 - 5 — `EMERGENCY`;
 - 6 — `OFFLINE`.
- ◆ `state_desc` — текстовое описание состояния БД: `ONLINE`, `RESTORING`, `RECOVERING`, `RECOVERY_PENDING`, `SUSPECT`, `EMERGENCY`, `OFFLINE`. Эта характеристика является, разумеется, производной от `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 БД, а также многие интересные характеристики этих файлов.

Наиболее важными для нас столбцами этого представления будут:

- ◆ `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` — 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 `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 Кб 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 объем всего дискового пространства или пока не достигнет допустимого предела (2 Тб для журнала транзакций или 16 Тб для файла данных);
 - положительное число указывает максимальный размер файла в страницах. Число 268 435 456 может быть указано только для файла журнала транзакций. Означает, что файл может расти до максимального размера в 2 Тб.
- ◆ `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*;
 - 5 — *OFFLINE*;
 - 6 — *DEFUNCT*.
- ◆ *state_desc* — текстовое описание состояния файла, производное от *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 Тб для журнала транзакций или 16 Тб 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 понадобятся вам при разработке БД. Давайте теперь очень кратко рассмотрим и некоторые другие средства, которые 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

Перечислим еще некоторые полезные системные представления:

- ◆ `sys.schemas` — возвращает сведения о схемах базы данных. Каждая БД может содержать более двух миллиардов схем. Нужно сказать, что объект "схема" (schema) в SQL Server, это совсем не то, что схема в некоторых других СУБД. О схемах мы поговорим в *разд. 3.8*.
- ◆ `sys.database_permissions` — возвращает сведения о полномочиях в БД. 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 принципах (владельцах или, иными словами, участниках доступа к БД и их объектам) в базе данных. Это опять же связано с вопросами безопасности в БД.
- ◆ `sys.database_role_members` — возвращает сведения о членах (участниках) ролей в БД. И роли базы данных при правильном их использовании могут быть хорошим средством для повышения безопасности БД.

Опишем другие системные представления просмотра каталогов для объектов, скажем так, "детального уровня":

- ◆ `sys.tables` — возвращает сведения о таблицах БД. Здесь даются сведения обо всех таблицах текущей базы данных.
- ◆ `sys.views` — возвращает сведения о представлениях в базе данных. Каждая БД 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. Индексы — это замечательный объект реляционной БД, который в одно и то же время может резко ухудшить работу с данными в БД, 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 БД. События — это очень интересное и полезное средство при работе с базами данных. Они позволяют синхронизировать работу программ, одновременно использующих одну и ту же БД, 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, предусмотрен оператор `EXECUTE`. Примеры таких обращений мы вскоре рассмотрим.

Перечислим распространенные хранимые процедуры:

`sp_databases` — выдает список баз данных.

`sp_stored_procedures` — возвращает список хранимых процедур.

`sp_help` — возвращает список различных объектов БД, 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 БД.

`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()`.

Получить идентификатор базы данных, который присваивается ей при ее создании, позволяет функция `DB_ID()`. Очень удобная и популярная 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 БД.

Функция `FILE_ID()` для текущей БД 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.

ЗАМЕЧАНИЕ

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 Transact-SQL `CREATE DATABASE`, так и диалоговых средств Management Studio. Кроме создания БД мы здесь научимся их удалять при помощи простого оператора `DROP DATABASE`, а также с помощью диалоговых средств Management Studio.

Для отображения характеристик и состояния существующей БД средствами Transact-SQL можно использовать описанные системные представления и диалоговые средства, имеющиеся 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

Создать новую базу данных позволяет оператор `CREATE DATABASE`. Его синтаксис (только для целей создания новой базы данных) показан в листинге 3.1. Основные синтаксические конструкции, как мы ранее договорились, будем представлять и в нотациях Бэкуса–Наура.

Листинг 3.1. Синтаксис оператора `CREATE DATABASE`. Вариант создания новой БД

```
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 БД, возможно, на другом компьютере к списку баз данных текущего экземпляра сервера, а также для создания так называемого мгновенного снимка БД (`SNAPSHOT`). Все это мы рассмотрим чуть позже в данной главе.

В результате успешного создания новой базы данных появятся все файлы БД, указанные в операторе или сформированные по умолчанию с соответствующими характеристиками. 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 БД обязательно нужно задать только логическое имя базы данных, которое будет известно в текущем экземпляре сервера. Все остальные конструкции необязательные. В этом случае самой базе данных и файлам создаваемой БД будут присвоены значения всех характеристик по умолчанию, которые мы с вами и рассмотрим в ближайшее время.

Логическое имя базы данных

Логическое имя базы данных — обязательный параметр. Оно идентифицирует создаваемую БД. Имя должно соответствовать правилам задания идентификаторов, обычных или с разделителями (см. главу 2). В обычном идентификаторе здесь также допустимы буквы кириллицы, если при инсталляции сервера был задан порядок сортировки `Cyrillic_General_CI_AS`. В идентификаторах с разделителями, как вы помните, возможны *любые* символы. Такой идентификатор нужно заключить в квадратные скобки. Имя не должно содержать более 128 символов. Оно должно быть уникальным среди имен всех баз данных текущего экземпляра SQL Server. По этому имени вы обращаетесь к конкретной БД 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 БД, в котором создается база данных. Если указано `NONE` (значение по умолчанию), то создается обычная (неавтономная) БД.

Если же указано значение `PARTIAL`, то создается так называемая *partially contained* база данных, т. е. частично автономная. Пользователь может подключаться к такой БД, используя отдельные средства аутентификации, не связанные с уровнем экземпляра сервера. Такую базу данных проще перенести в другой экземпляр сервера

БД, на другой компьютер. Кроме того, в таких БД 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 на примерах.

Предложение **ON**

```
[ ON [ PRIMARY ] [ <спецификация файла> [, <спецификация файла>]...  
[, <файловая группа> [, <файловая группа>] ...]]]
```

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 мы поговорим далее.

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

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

Предложение **LOG ON**

```
[ LOG ON <спецификация файла> [, <спецификация файла> ]... ]
```

Необязательное предложение **LOG ON** описывает файл (файлы) журнала транзакций. Оно может присутствовать только при задании предложения **ON**. В базе данных может существовать большое число 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 БД вы не укажете файл данных или файл журнала транзакций, то система всем характеристикам этих файлов присвоит значения по умолчанию.

Предложение **COLLATE**

```
[ COLLATE <порядок сортировки> ]
```

Необязательное предложение **COLLATE** позволяет задать для создаваемой базы данных порядок сортировки, отличный от того, который был установлен при инстал-

ляции системы для экземпляра сервера. Если предложение не указано, то БД будет иметь порядок сортировки по умолчанию, заданный при установке системы. При установке SQL Server по умолчанию был установлен порядок сортировки Cyrillic_General_CI_AS.

Предложение **WITH**

[WITH <опция> [, <опция>]...]

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

<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 }
}
```

Файловый поток

После ключевого слова FILESTREAM в скобках перечисляются опции файлового потока:

<опция файлового потока> ::=

```
{ NON_TRANSACTED_ACCESS = { OFF | READ_ONLY | FULL }
| DIRECTORY_NAME = '<имя каталога>' }
```

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

- ◆ 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*.

Другие опции предложения **WITH**

- ◆ DEFAULT_FULLTEXT_LANGUAGE (допустимо только для автономной БД) — 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 приведен в *приложении 5*.

- ◆ `DEFAULT_LANGUAGE` (допустимо только для автономной БД) — задает язык по умолчанию для вновь создаваемых регистрационных имен пользователей. Может задаваться в виде кода языка, его названия или псевдонима.
- ◆ `NESTED_TRIGGERS` (допустимо только для автономной БД) — задает возможность использования вложенных триггеров `AFTER`. Если указано `OFF`, вложенные триггеры недопустимы. При задании `ON` может существовать до 32 уровней триггеров. Иными словами, триггер может вызывать (разумеется, неявно) другой триггер, который, в свою очередь, инициирует обращение к триггеру следующего уровня. И так до 32 уровней.
- ◆ `TRANSFORM_NOISE_WORDS` (допустимо только для автономной БД) — задает поведение сервера БД 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 число строк, то просто выдается предупреждающее сообщение.

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

- ◆ `TWO_DIGIT_YEAR_CUTOFF` (допустимо только для автономной БД) — 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 выбрано столетие, следующее за указанным в операторе столетием. Хорошая практика — указание во *всех* датах четырехсимвольного значения года.
- ◆ `DB_CHAINING` — определяет, может ли создаваемая БД находиться в цепочках связей между несколькими базами данных. `OFF` (по умолчанию) запрещает такое использование, `ON` — разрешает.
- ◆ `TRUSTWORTHY` — задает условие, могут ли программные компоненты базы данных (хранимые процедуры, представления, созданные пользователем функции) обращаться к ресурсам вне БД. `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*.

Листинг 3.2. Синтаксис спецификации файла

```

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

```

В спецификации файла обязательны только предложения NAME (логическое имя файла) и FILENAME (имя файла в операционной системе). При отсутствии любого из предложений SIZE, MAXSIZE или FILEGROWTH соответствующим характеристикам будут присвоены значения по умолчанию, которые мы рассмотрим далее. Эти значения по умолчанию различны для файлов данных и для файлов журнала транзакций.

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 БД не описывается (отсутствует предложение ON), то логическому имени единственного файла данных присваивается имя самой базы данных. Например, если создаваемая БД 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 необязательны, 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 позво-

лит восстановить базу 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 при установке 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 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 установке системы. В нашем случае это будет путь

```
C:\Program Files\Microsoft SQL Server\MSSQL16.MSSQLSERVER\MSSQL\DATA
```

ЗАМЕЧАНИЕ

Пути по умолчанию для файла данных и файла журнала транзакций можно изменить в Management Studio. Для этого в **Обозревателе объектов** нужно щелкнуть правой кнопкой мыши по имени сервера базы данных и в контекстном меню выбрать элемент **Свойства**. В появившемся окне свойств сервера нужно выбрать вкладку **Параметры базы данных** и изменить пути в полях **Данные** и **Журнал** 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 не совсем понятной для меня причине недопустимо присутствие каких-либо выражений, внутренних переменных, операций. В частности, нельзя даже задать 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 есть 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 мы рассмотрим далее на примерах. Кроме того, в утилите командной строки `sqlcmd` существует возможность вызывать на выполнение скрипт, содержащий параметры, значения которым подставляются при вызове этой утилиты. Такой пример мы также вскоре рассмотрим.

В случае, когда спецификация файла задает файл файлового потока (`filesystem`), путь к файлу указывают только имена вложенных 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. Параметр **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 задана, то предполагаются мегабайты. Обратите внимание, что в описании синтаксиса значение **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 Кб, а первичный файл данных должен иметь размер не менее 3 Мб. Параметр **целое1** имеет целочисленный тип данных **INTEGER**, следовательно, его значение не может превышать 2 147 483 647. 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 такую проверку.

Если предложение **SIZE** не задано, то начальному размеру файла присваивается значение по умолчанию, определенное в системной базе данных **model**. В моей версии системы файл данных получает размер 3 Мб, а журнал транзакций — 1 Мб.

Предложение **MAXSIZE**

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

Необязательное предложение **MAXSIZE** задает максимальный размер, который может получить файл при увеличении объема 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, а файл журнала транзакций — 2 терабайта. Как и при задании начального размера файла в этом предложении максимальный размер можно указать в килобайтах, мегабайтах, гигабайтах и терабайтах (параметры **KB**, **MB**, **GB** и **TB** соответственно).

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 прира-

щения не указана, то принимается мегабайт. Если это предложение вообще отсутствует, то для файла данных приращение задается в 1 Мб, а для журнала транзакций устанавливается приращение в 10% от начального размера файла.

Размер страницы файла данных в базе имеет фиксированное значение 8 Кб (8192 байта) и не может быть изменен ни при создании базы данных, ни при ее изменении.

Файловая группа

Синтаксис описания файловой группы приведен в *листинге 3.3*.

Листинг 3.3. Синтаксис описания файловой группы

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

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.

Необязательное предложение `CONTAINS MEMORY_OPTIMIZED_DATA` означает, что файловая группа хранит данные, оптимизированные в памяти. В базе данных может присутствовать только одна файловая группа `MEMORY_OPTIMIZED_DATA`.

Средства таблиц, оптимизированных в памяти (In-Memory OLTP), появились в SQL Server 2014. Они позволяют во многих случаях сильно повысить производительность системы при использовании коротких транзакций.

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 создания новой базы данных (другие варианты использования этого оператора мы рассмотрим ближе к концу главы) не позволяет явно установить начальные значения тому множеству характеристик БД, ее файловых групп и файлов, которые существуют в диалоговых средствах SQL Server. Для изменения значений характеристик по умолчанию предусмотрен оператор `ALTER DATABASE`. Его мы очень скоро будем рассматривать довольно подробно. В диалоговых средствах Management Studio и при первоначальном создании базы данных можно задавать значения для большинства характеристик.

3.4.1.2. Оператор удаления базы данных

Для удаления созданной пользователем в текущем экземпляре сервера базы данных предназначен оператор `DROP DATABASE`. Его синтаксис показан в листинге 3.4.

Листинг 3.4. Синтаксис оператора `DROP DATABASE`

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

Оператор `DROP DATABASE` позволяет удалить одну или более указанных пользовательских баз данных или мгновенных снимков БД (см. *раздел 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 БД никак не влияет на базу данных-источник.

При удалении БД она удаляется из списка баз данных экземпляра сервера. Также с внешних носителей физически удаляются все файлы, относящиеся к этой БД, — 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 БД неактивна (находится в состоянии `OFFLINE`), то файлы не удаляются.

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

Далее 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 БД, 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.

Универсальная утилита командной строки, позволяющая выполнять операторы `Transact-SQL` в `SQL Server`, — `sqlcmd`. Описание наиболее часто используемых параметров этой утилиты приведено в *приложении 3*.

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

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

В подсказке этого окна введите имя утилиты — `sqlcmd`. Если на вашем компьютере установлен единственный экземпляр сервера БД 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 имя экземпляра сервера:

```
sqlcmd -S DESKTOP-5CF26M3
```

На экране появится подсказка самой утилиты:

```
1>
```

Вначале отобразим существующие в экземпляре сервера БД, используя системное представление `sys.databases`. Введите в строке подсказки утилиты вначале команду `USE`, указывающую, что текущей базой данных является системная БД `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 БД в системном представлении `sys.databases`

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

На экране будут отображаться введенные операторы:

```
1> USE master;  
2> GO  
Контекст базы данных изменен на "master".  
1> SELECT name, database_id, create_date, collation_name  
2> FROM sys.databases;  
3> GO
```

Появится список всех баз данных, существующих в экземпляре сервера. Если вы еще не создавали пользовательские БД, то список будет содержать только описания четырех системных баз данных: **master**, **tempdb**, **model** и **msdb**. Скрытая база данных **resource** не отображается в этом списке.

Оператор `SELECT` позволяет в этом примере получить указанные данные из таблицы (таблиц) базы данных или из представления. Он также дает возможность 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 мы получим список всех баз данных, существующих в текущем экземпляре сервера БД.

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 текста. И дело не только в том, что по умолчанию длина командной строки составляет 80 символов. Размер строки можно изменить. Неприятность в том, что размер отображаемых полей слишком велик.

Чтобы улучшить восприятие выводимой информации 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 БД 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 он выглядит следующим образом:

NAME	ID	DATE	COLLATION
master	1	2003-04-08 09:13:36.390	Cyrillic_General_CI_AS
tempdb	2	2023-02-22 09:07:20.807	Cyrillic_General_CI_AS
model	3	2003-04-08 09:13:36.390	Cyrillic_General_CI_AS
msdb	4	2022-02-20 20:49:38.857	Cyrillic_General_CI_AS

(обработано строк: 4)

В операторе для трех столбцов присутствует весьма простая и очень удобная в работе функция 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 эта функция у нас лишь уменьшает размер 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 числом символов 128 опять же в строковый, но с другим размером, указав в выходном типе данных 20 символов (`CHAR(20)`).

Для столбца, содержащего дату и время, мы в операторе `SELECT` не задали никакого преобразования, 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 длина таких заголовков будет несколько больше. 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 не заключать в апострофы. В некоторых информационных и многих учебных материалах корпорации Microsoft, да и в документе Books Online, вы можете увидеть примеры такого синтаксиса. Такая свобода задания строковых значений в данном случае, в общем-то, совершенно понятна. В этом месте ожидается наличие строкового данного, и соответствующий набор знаков воспринимается как строка символов. Наверное, не стоит напоминать, что это не является хорошей практикой. Если в наших пакетах есть строковые константы (в данном случае при задании заголовков), то всегда следует заключать их в апострофы, чтобы избежать возможной путаницы, повысить читаемость кода и не получить лишних неприятностей при изменениях системы в будущем.

Несколько слов об операторе `GO`. Он не является оператором языка Transact-SQL, поэтому не может завершаться символом точка с запятой, наличие этого символа вызовет синтаксическую ошибку. Это служебный оператор (или команда) 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 [<число повторений>]
```

Если число повторений (которое должно быть положительным целым числом) задано, то весь предыдущий фрагмент пакета выполняется заданное число раз. Если число повторений не указано, то пакет выполняется один раз. Выполните предыдущий пакет операторов, указав оператор `GO` с любым числом повторений, большим единицы. Вы получите заданное вами число 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 БД 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	2023-02-22 09:07:20.807	Cyrillic_General_CI_AS

(обработано строк: 1)

ЗАМЕЧАНИЕ

В самом начале каждого пакета мы записывали оператор `USE`, который указывает текущую БД, 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 БД является текущей. Оператор `USE` в этих примерах можно опустить. Во многих других ситуациях при применении системных средств работы с базами данных важно, какая БД 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 программных средствах и вообще об удобстве в работе. Утилиту `sqlcmd` можно запускать на выполнение в обычной командной строке.

Позже мы сможем сравнить вид этого отображения с тем, что можно получить в программе `Management Studio`. Нужно 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 БД можно получить через 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 БД 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 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 БД 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	ONLINE	760		-1 10
1	LOG	mastlog	ONLINE	288		-1 10
2	ROWS	tempdev	ONLINE	1024		-1 10

2	LOG	templog	ONLINE 1024	-1 10
3	ROWS	modeldev	ONLINE 1024	-1 128
3	LOG	modellog	ONLINE 160	-1 10
4	ROWS	MSDBData	ONLINE 1992	-1 10
4	LOG	MSDBLog	ONLINE 1024	268435456 10

(обработано строк: 8)

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, вызвав 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 БД 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
tempdb	tempdev	ONLINE	ROWS
tempdb	templog	ONLINE	LOG

(обработано строк: 8)

Благодаря наличию 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. В последней строке 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. Такое возможно в MS SQL Server.

Предложение `ORDER BY` можно задать и в виде `ORDER BY 1`. Здесь указывается, что упорядочение осуществляется по первому полю из списка выбора. Поля нумеруются, начиная с единицы. 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`.

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

Пример 3.7. Отображение файлов БД 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	760
2	1	LOG	mastlog	0	ONLINE	288

(обработано строк: 2)

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

Пример 3.8. Отображение файлов БД 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 БД 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 БД master и подставить это значение в предложение `WHERE`.

На самом деле здесь можно и в одном операторе осуществить поиск идентификатора нужной базы данных, добавив 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 БД master — вызов системной функции `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`.

Введите и выполните операторы *примера 3.9*.

Пример 3.9. Лучший вариант отображения файлов БД 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, 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 БД master. Однако если в системе (в текущем экземпляре сервера БД) 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 намного 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 Бд. 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, ошибки не возникнет.

Выполните следующие операторы (пример 3.10).

Пример 3.10. Отображение логических имен файлов БД 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
```

Результат:

ID	Файл 1	Файл 2	Ничего
1	master	mastlog	NULL

Здесь в операторе `USE` указывается БД `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 и идентификатор текущей базы данных. В функции `DB_ID()` мы не указали никакого имени базы данных, поэтому функция вернет идентификатор текущей Бд, 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 предположение `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), откуда утилита будет читать операторы. Можно поместить пакет операторов в файл, корректировать многократно и выполнять при вызове утилиты `sqlcmd`. Пример вызова файла скрипта:

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

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

Еще про один параметр `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 чувствительно к регистру: вы должны ввести именно строчную букву `o`, а не прописную.

Пример указания 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 данных, существующих в файле.

Описание наиболее полезных параметров утилиты sqlcmd содержится в *приложении 2*.

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

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

Пример 3.11. Создание и отображение БД 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	1024
2	1	LOG	SimpleDB_log	0	ONLINE	1024

Чаще всего в процессе проектирования баз данных вы будете создавать БД 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 выпол-

нение. Как правило, база данных пересоздается вами заново. Если вы забудете 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:

Сообщение 1801, уровень 16, состояние 3, сервер DESKTOP-5CF26M3, 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 допустимы операторы ветвления, в частности, оператор `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" БД

```
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 БД, 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, циклов и некоторых других в пакетах SQL Server — необыкновенно удобное средство. Не все СУБД 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 БД — 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 БД установлен начальный размер 5 Мб, приращение указано 1 Мб, максимальный размер не ограничивается, следовательно, файл может расти до исчерпания объема дискового пространства или до 16 Тб.

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

МАЛЕНЬКОЕ ЗАМЕЧАНИЕ ПО СИНТАКСИСУ

При указании размеров в операторе `CREATE DATABASE` единицы измерения можно записывать сразу же после числа, а можно помещать через один или более пробелов. Во втором случае запись выглядит, как мне кажется, намного 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 возможны значения по умолчанию, попробуйте разобраться в скриптах, написанных для других СУБД, где заданы принятые именно *там* значения по умолчанию.

Следующий пример в принципе повторяет предыдущий, однако здесь мне хочется рассмотреть некоторые дополнительные полезные средства, имеющиеся 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 применяем 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 объявленные 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` (его имя можно сократить до `EXEC`), которому в качестве параметра передаем строку, создан-

ную опять же при выполнении конкатенации строковых констант и значений локальных параметров @path_data и @path_log. Вся строка заключена 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 заменить оператор EXECUTE на SELECT. Строка 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 не просто демонстрирует некоторые возможности, существующие 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 БД

```
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 нецелесообразно. 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 БД. База данных помимо первичной файловой группы может содержать и произвольное число других файловых групп, называемых пользовательскими, или вторичными, файловыми группами.

Теперь создадим базу данных с двумя файловыми группами, 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 ]
  [ CONTAINS MEMORY_OPTIMIZED_DATA ]
  <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 БД 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 БД 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

(обработано строк: 6)

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

Теперь отобразите сведения только по файловым группам, применив системное представление `sys.filegroups`, как 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 'Description',
       CAST(is_read_only AS CHAR(1)) AS 'Read-only'
FROM sys.filegroups;
GO
```

Результат:

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

(обработано строк: 2)

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

Завершая рассмотрение утилиты командной строки `sqlcmd`, мне хочется показать вам одну возможность использования 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 в параметре утилиты `-i`.

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

```
$(имя параметра>)
```

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

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

Создайте, например в Блокноте, скрипт, приведенный в *примере 3.19*.

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 `${DBNM}` служит для задания имени базы данных, имени каталога на диске D: для размещения файлов создаваемой базы данных, для задания логических имен файла данных и файла журнала транзакций и для задания имен файлов в операционной системе. При задании путей к файлам, логических и физических имен файлов используется конкатенация, соединение нескольких строк. Причем никаких знаков операции для конкатенации в этом случае не требуется. Параметр `${DBNM}` просто будет заменен значением, заданным при вызове утилиты.

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:

```

Контекст базы данных изменен на "master".
ID Type Description Name          Size
-- ----

```

```
1 0    ROWS      DBParam_dat  640
2 1    LOG       DBParam_log   256
```

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;
```

Аналогично 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 число параметров. Синтаксис задания значений при вызове утилиты довольно свободный. 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 присутствуют 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, 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, вы легко можете их изменять мышью.

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 — очень мощное и удобное средство работы с базами данных в SQL Server.

Запустите на выполнение Management Studio. Щелкните мышью по кнопке **Пуск**, затем по строке **Microsoft SQL Server Tools 19** и наконец по элементу **SQL Server Management Studio**. Появится диалоговое окно соединения с сервером. В окне **Соединение с сервером** можно выбрать тип сервера — выпадающий список **Тип сервера**, имя сервера (имя одного из экземпляров сервера, установленных на компьютере — если существует несколько серверов), вид аутентификации **Проверка подлинности** (выберите из выпадающего списка элемент **Проверка подлинности Windows**) и ввести имя пользователя (*рис. 3.3*).

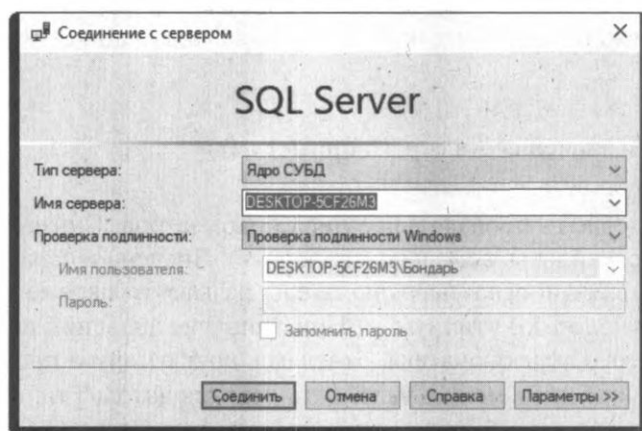


Рис. 3.3. Соединение с сервером

Если на компьютере установлен только один экземпляр сервера БД, то в этом окне нужно, ничего ни изменяя и не выбирая, просто щелкнуть по кнопке **Соединить**. Произойдет соединение с текущим экземпляром Database Engine и следом появится главное окно Management Studio, показанное на *рис. 3.4*.

Однако если на вашем компьютере установлено несколько серверов БД, то вам нужно из выпадающего списка **Имя сервера** 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.

Если в окне не виден **Обозреватель объектов**, выберите в меню элементы **Вид | Обозреватель объектов** 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 **Обозреватель объектов**, при помощи которого вы можете получить быстрый доступ ко многим полезным возможностям Management Studio, большинство из которых мы вскоре рассмотрим.

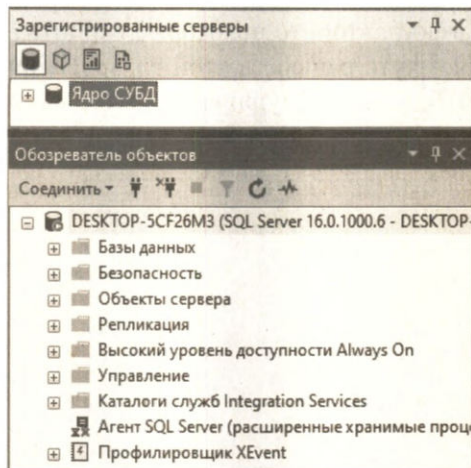


Рис. 3.4. Главное окно Management Studio.
Панель **Обозреватель объектов**

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

Для вызова этого окна можно также щелкнуть мышью по элементу меню **Файл**, выбрать элемент **Создать**, а затем элемент **Запрос в текущем соединении**. В дальнейшем подобный вызов элемента в многоуровневом меню мы будем изображать в тексте чуть короче в следующем виде: "Выберите в меню элементы **Файл | Создать | Запрос в текущем соединении**".

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. 3.5.

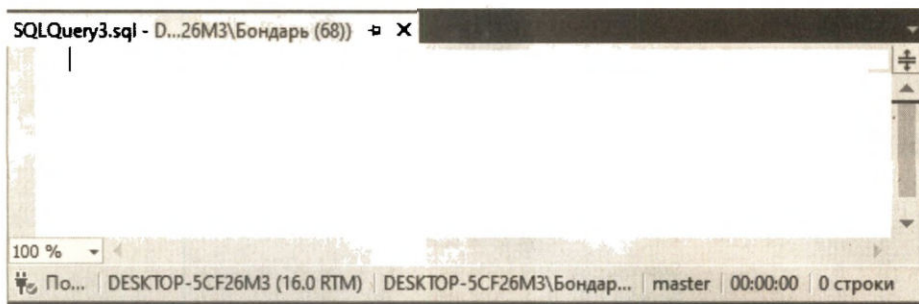


Рис. 3.5. Окно ввода операторов Transact-SQL

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 БД 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 выделяет цветом ключевые слова, константы, 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 введенных операторов щелкните мышью по кнопке **Выполнить** на панели инструментов, выберите в меню элементы **Запрос | Выполнить** либо нажмите клавишу <F5> или клавиши <Ctrl>+<E>.

Результат выполнения пакета *примера 3.20* показан на *рис. 3.6*. Это окно появится в нижней части окна выполнения запросов.

	ID	Type	Description	Name	State	State desc	Size
1	1	0	ROWS	SimpleDB	0	ONLINE	1024
2	2	1	LOG	SimpleDB_log	0	ONLINE	1024

Рис. 3.6. 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, по-

казанного на рис 3.9, не нужно рассчитывать количество символов, которое уместится на выходе. Нет необходимости преобразовывать отображаемые столбцы 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 подходящего варианта. Для этого нужно курсор мыши подвести к границе двух столбцов в заголовке и, нажав левую кнопку, изменять требуемый размер. Как правило, программа Management Studio с самого начала предоставляет хорошо читаемый текст.

Вы можете заранее любыми средствами (пусть даже в Блокноте) 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 элементы **Файл | Открыть | Файл**. Можно нажать клавиши `<Ctrl>+<O>` или щелкнуть мышью по кнопке открытия файла 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 элементы **Файл | Сохранить (имя скрипта)**, нажав клавиши `<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 элементы **Файл | Сохранить [имя скрипта] как ...** 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 от Microsoft рассказывали что-нибудь интересное о возможностях системы с демонстрацией этих возможностей при помощи Management Studio, то вы, конечно же, видели, что они постоянно пользовались этим приемом.

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 в операторе `SELECT` и системном представлении `sys.master_files`. Щелкните мышью по символу + слева от строки **Базы данных** в окне **Обозреватель объектов**. Раскроется список баз данных, определенных в текущем экземпляре сервера БД (рис. 3.7). Имена баз данных в списке упорядочены по алфавиту. Только хочу напомнить, что те средства, с которыми вы работали в утилите `sqlcmd` для отображения характеристик баз данных и их файлов, можно с тем же успехом задействовать в Management Studio.

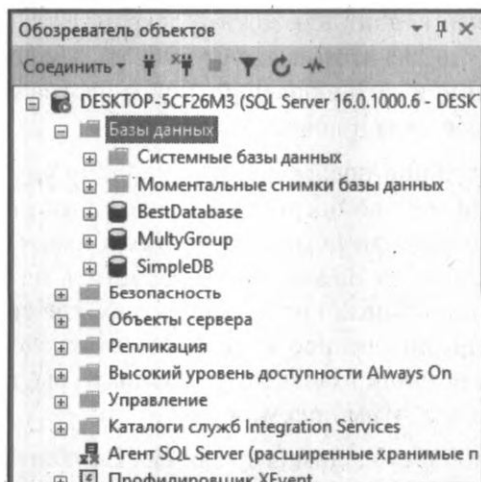


Рис. 3.7. Список баз данных в Обозревателе объектов

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

ЗАМЕЧАНИЕ

Если вновь созданная база данных в окне **Обозреватель объектов** не видна (а скорее всего так и будет сразу после создания новой БД), то следует обновить список объектов, щелкнув правой кнопкой мыши по строке **Базы данных** или по имени сервера базы данных и выбрав в появившемся контекстном меню элемент **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 **Обозреватель объектов**, щелкнув мышью по заголовку этого окна, и выделить мышью строку **Базы данных** 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 контекстном меню выберите элемент **Свойства**. Появится окно свойств выбранной БД, где текущей является вкладка **Общие** (рис. 3.8).

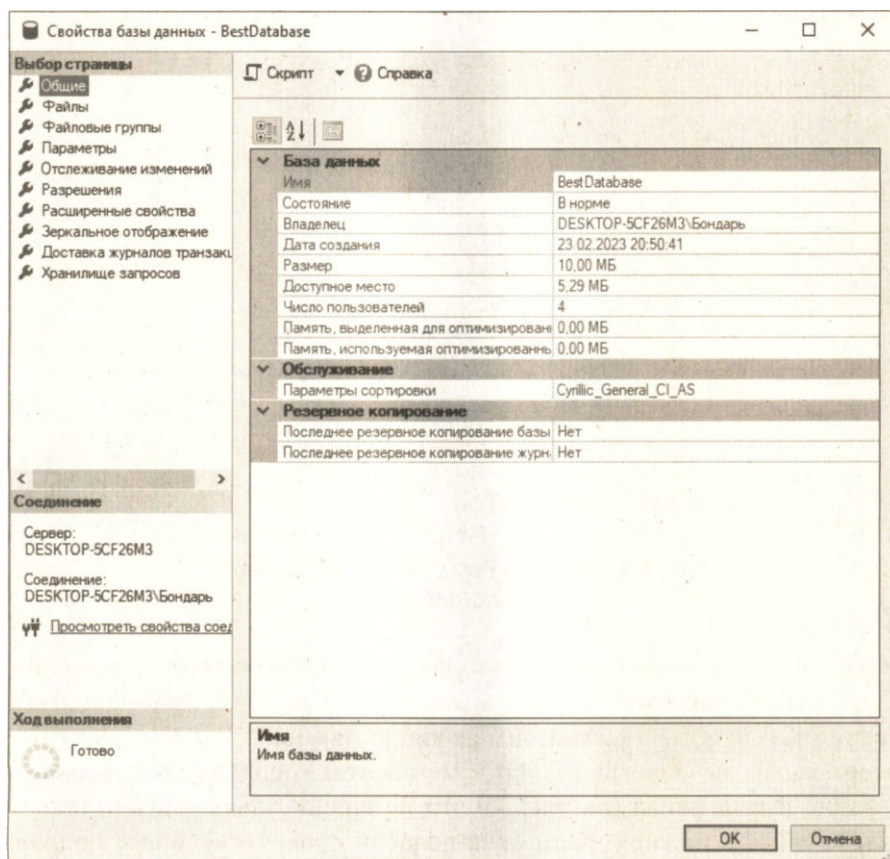


Рис. 3.8. Свойства базы данных BestDatabase. Вкладка **Общие**

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

В левой верхней части окна, в панели **Выбор страницы**, щелкните мышью по строке **Файлы**. Появится более интересная вкладка (рис. 3.9), где присутствует описание характеристик всех файлов, входящих в состав этой базы данных.

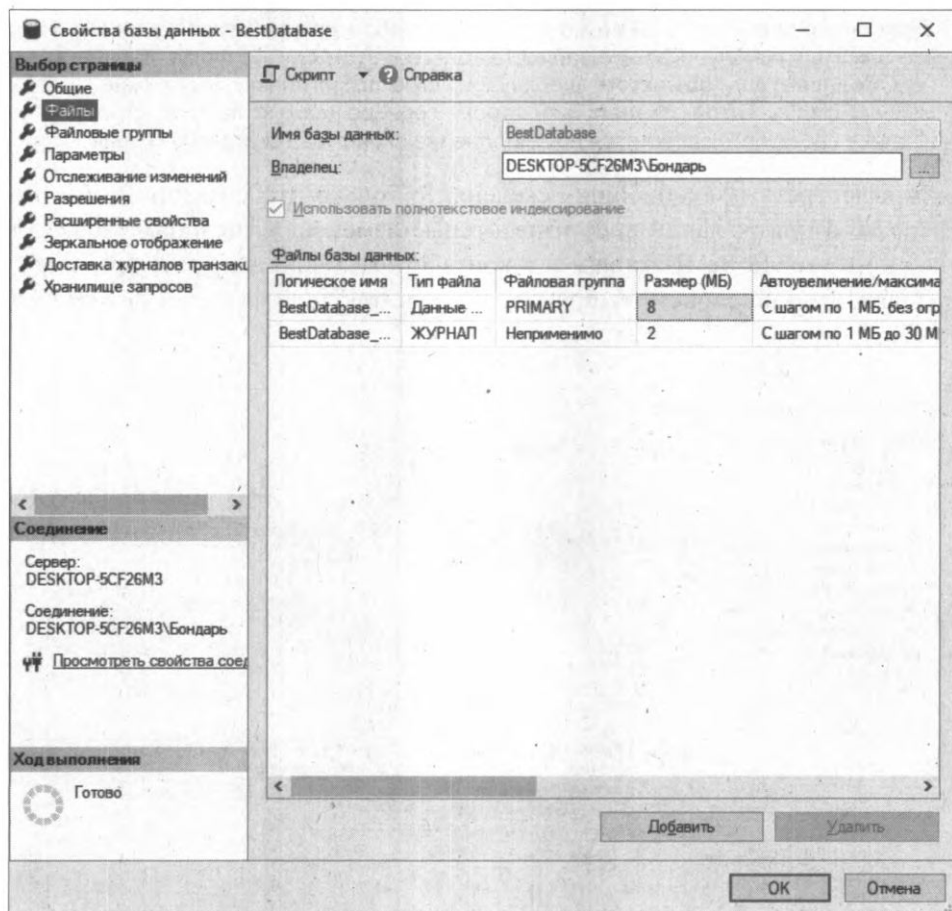


Рис. 3.9. Свойства базы данных BestDatabase.
Вкладка **Файлы**

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: **Данные СТРОК** и **ЖУРНАЛ**. Для них представлены начальный размер в мегабайтах, порядок увеличения размера, путь к файлу и имя файла (на рис. 3.9 это не видно, однако можно просмотреть, воспользовавшись в нижней части окна полосой прокрутки). Более подробно отображаемые характеристики файлов при полностью развернутом окне свойств БД показаны на рис. 3.10.

Файлы базы данных:						
Логическое имя	Тип файла	Файловая группа	Размер (МБ)	Автоматическое увеличение/максимальный размер	Путь	Имя файла
BestDatabase_dat	Данные СТРОК	PRIMARY	8	С шагом по 1 МБ, без ограничений	D:\BestDatabase	Winner.mdf
BestDatabase_log	ЖУРНАЛ	Неприменимо	2	С шагом по 1 МБ до 30 МБ	D:\BestDatabase	Winner.ldf

Рис. 3.10. Характеристики файлов
базы данных BestDatabase

Давайте подробнее рассмотрим этот список.

В столбце **Логическое имя** мы видим логические имена двух созданных файлов — файла данных `BestDatabase_dat` и файла журнала транзакций `BestDatabase_log`.

В столбце **Тип файла** указывается именно тип файла — **Данные СТРОК** или файл журнала транзакций (**ЖУРНАЛ**).

Столбец **Файловая группа** 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 содержится текст **Неприменимо**, поскольку для файлов журнала транзакций не применяется распределение по файловым группам.

Столбец **Размер (МБ)** указывает размер файла в мегабайтах.

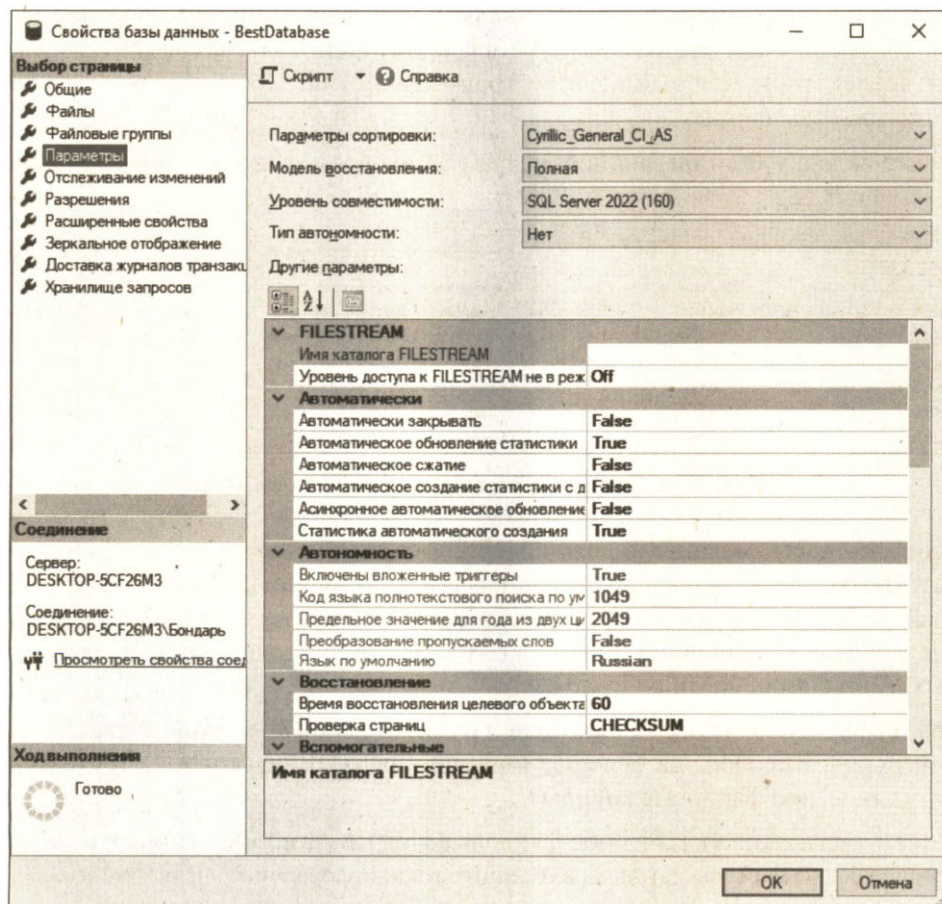


Рис. 3.11. Свойства базы данных BestDatabase. Вкладка **Параметры**

В столбце **Автоувеличение/максимальный размер** описывается единица увеличения размера памяти (единица измерения или проценты от начального значения) и

величина, до которой может увеличиваться размер памяти, отводимой под файл, либо указывается, что размер не ограничивается (**без ограничений**).

Столбец **Путь** содержит полный путь к файлу.

В столбце **Имя файла** содержится имя физического файла в операционной системе.

При выборе в панели **Выбор страницы** вкладки **Параметры** появится следующее окно (рис. 3.11).

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

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 **Отмена** в любой вкладке этого окна. Щелкните правой кнопкой мыши по имени базы данных MultyGroup в **Обозревателе объектов** и в появившемся контекстном меню выберите строку **Свойства**. Появится окно просмотра свойств этой базы данных.

Выберите вкладку **Файлы** для просмотра файлов базы данных. Список файлов показан на рис. 3.12.

Логическое имя	Тип файла	Файловая группа	Размер (МБ)	Автоувеличение/максимальный размер	Путь	Имя файла
MultyGroup1	Данные СТРОК	PRIMARY	8	С шагом по 64 МБ, без ограничений	...	D:\MultyGroup\MultyGroup1.mdf
MultyGroup3	Данные СТРОК	MultyGroup2	8	С шагом по 64 МБ, без ограничений	...	D:\MultyGroup\MultyGroup3.ndf
MultyGroup4	Данные СТРОК	MultyGroup2	8	С шагом по 64 МБ, без ограничений	...	D:\MultyGroup\MultyGroup4.ndf
MultyGroup2	Данные СТРОК	PRIMARY	8	С шагом по 64 МБ, без ограничений	...	D:\MultyGroup\MultyGroup2.ndf
MultyGroupLog1	ЖУРНАЛ	Неприменимо	8	С шагом по 64 МБ до 2097152 МБ	...	D:\MultyGroup\MultyGroupLog1.ldf
MultyGroupLog2	ЖУРНАЛ	Неприменимо	8	С шагом по 64 МБ до 2097152 МБ	...	D:\MultyGroup\MultyGroupLog2.ldf

Рис. 3.12. 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 выберите вкладку **Файловые группы**. Данные в этой вкладке весьма скромненькие. 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 **По умолчанию**, 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 БД.

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

Замечания по использованию Management Studio

Напомню, что в Management Studio есть удобная возможность в окне выполнения запросов вводить произвольное количество операторов, а для выполнения одного оператора или группы операторов нужно выделить необходимые строки и нажать клавишу <F5> или клавиши <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 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. Для этого перед выполнением группы операторов нужно в меню выбрать элементы **Запрос | Отправить результаты в | В виде текста** или нажать клавиши <Ctrl>+<T>.

Чтобы вернуться к более привычной форме отображения результатов в табличном виде, нужно выбрать в меню элементы **Запрос | Отправить результаты в | В виде сетки** или нажать клавиши <Ctrl>+<D>. Чтобы в меню появился элемент **Запрос**, нужно мышью сделать текущим окно запросов.

Кроме рассмотренных в Management Studio существует еще множество дополнительных настроек. Для получения доступа к многочисленным настройкам выберите в меню элементы **Запрос | Параметры запроса** или щелкните мышью по кнопке **Параметры запроса** на инструментальной панели. Появится окно параметров, в котором в нескольких вкладках можно установить большое количество характеристик, используемых при выполнении программы. Например, во вкладке **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, во вкладке **Общие** можно указать другой оператор, просто введя его текст в соответствующее поле, что, конечно же, никак нельзя порекомендовать.

Можно указать, что в окне запроса должны отображаться номера строк. Для этого нужно выбрать в меню элементы **Сервис | Параметры**. Появится диалоговое окно задания режимов. В левой части окна нужно раскрыть элемент **Transact-SQL** и щелкнуть мышью по элементу **Общие**. В правой части окна нужно отметить флажок **Нумерация строк** и щелкнуть по кнопке **ОК**.

После этой установки в левой части окна запроса будут выводиться номера строк. Это полезная возможность, особенно, когда вы отлаживаете достаточно "длинный" запрос, содержащий большое число 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, встречаются ситуации, когда подчеркиваются и совершенно верные тексты. Но это бывает достаточно редко.

3.4.2. Создание базы данных с использованием диалоговых средств Management Studio

Базу данных можно создавать не только с использованием оператора Transact-SQL `CREATE DATABASE`. В Management Studio существуют диалоговые средства, позволяющие задать все необходимые характеристики создаваемой БД.

Щелкните правой кнопкой мыши в **Обозревателе объектов** по строке **Базы данных** и в появившемся контекстном меню выберите элемент **Создать базу данных** (можно также щелкнуть правой кнопкой мыши по имени *любой* пользовательской базы данных и выбрать строку **Создать базу данных**). Появится окно создания новой базы данных (рис. 3.13), где можно создавать базу данных, ее файлы со всеми необходимыми характеристиками. Текущей будет вкладка **Общие**.

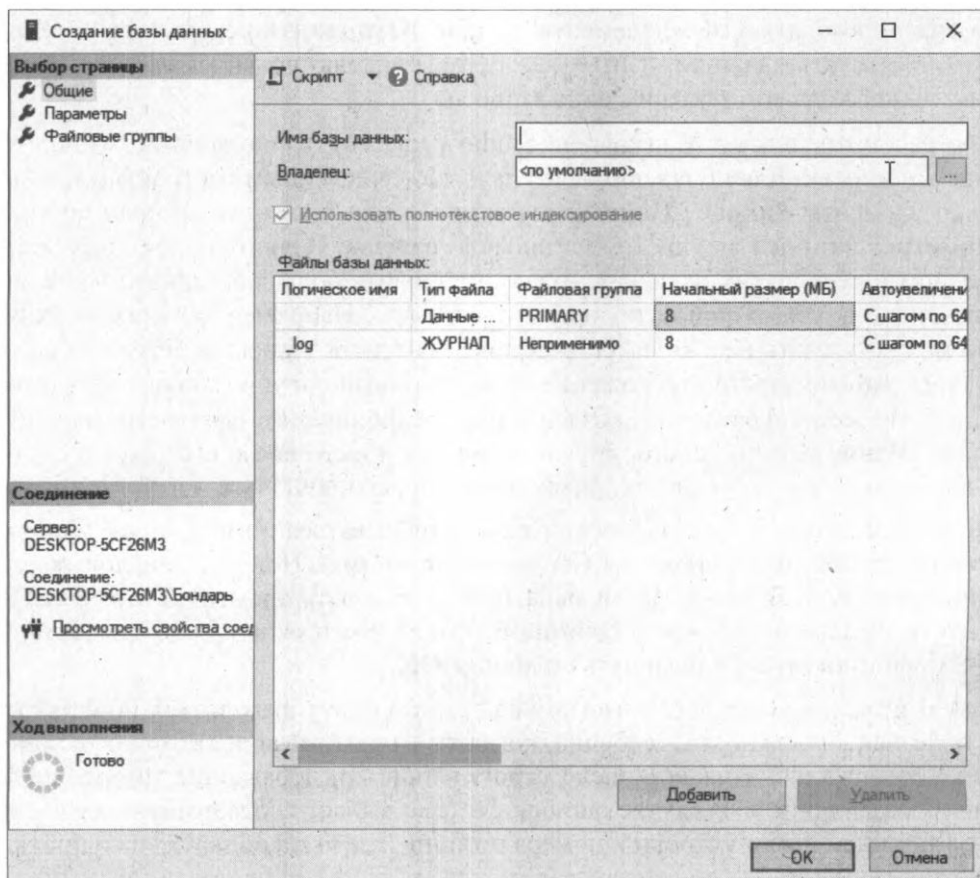



Рис. 3.13. Окно создания новой базы данных

Вначале зададим имя базы данных. Пусть это будет БД с простым и понятным именем **NewDatabase**. Введите это имя в поле **Имя базы данных** в верхней части окна. В процессе ввода имени в строках столбца **Логическое имя** 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 **Путь** (пути к файлам базы данных) программа предлагает нам некоторый вариант размещения файлов создаваемой БД. В правой части строки в этом поле присутствует кнопка с многоточием . Для выбора другого размещения файлов щелкните по этой кнопке. Откроется окно выбора папки — **Поиск папки** (рис. 3.14). Выберите по очереди для каждого из файлов тот же самый каталог, который был задан у нас для базы данных BestDatabase: D:\BestDatabase.

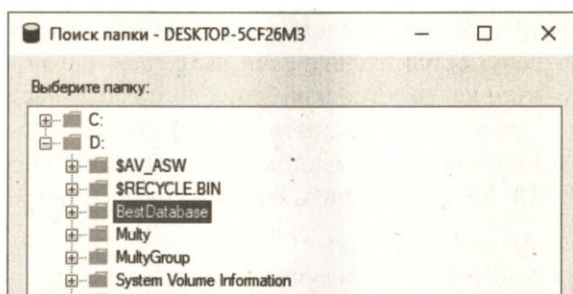


Рис. 3.14. Выбор папки для размещения файла базы данных

Щелкните в этом окне по кнопке **ОК**.

Каталог для размещения файлов можно задать и путем ввода в поля столбца **Путь** нужного пути к файлам, просто набрав с клавиатуры текст D:\BestDatabase.

В столбце **Имя файла** введите имена файлов, соответственно для файла данных и файла журнала транзакций: **NewDatabase.mdf** и **NewDatabase.ldf**.

Список файлов будет выглядеть как на рис. 3.15.

Файлы базы данных:						
Логическое имя	Тип файла	Файловая группа	Н...	Автоувеличение/максимальный размер	Путь	Имя файла
NewDatabase_dat	Данные СТРОК	PRIMARY	8	С шагом по 64 МБ, без ограничений	D:\BestDatabase	NewDatabase.mdf
NewDatabase_log	ЖУРНАЛ	Неприменимо	8	С шагом по 64 МБ, без ограничений	D:\BestDatabase	NewDatabase.ldf

Рис. 3.15. Файлы создаваемой базы данных

Для создания этой базы данных нужно щелкнуть по кнопке **ОК**, 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 **Добавить** в нижней части окна и добавить любое число файлов данных или файлов журналов транзакций.

Для этой базы данных создадим еще один файл данных с именем **NewDatabase2**. Щелкните по кнопке **Добавить**. 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, что и для других файлов этой базы данных, задайте для него и имя физического файла **NewDatabase2.ndf**.

Тип файла по умолчанию устанавливается как файл данных. Если вы собираетесь создавать файл журнала транзакций, то нужно в поле **Тип файла** щелкнуть мышью по тексту, и справа от значения этого поля появится кнопка со стрелкой вниз. При щелчке по кнопке появится выпадающий список, в котором вы в этом случае должны были бы выбрать значение **ЖУРНАЛ**. Сейчас оставим для типа файла значение по умолчанию.

Аналогично 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 **Файловая группа** и из выпадающего списка выбрать нужную файловую группу. Поскольку никаких вторичных файловых групп мы пока не создавали, список будет содержать только две строки: **PRIMARY** и **<создать файловую группу>**.

Создадим новую файловую группу этим способом. Другой путь создания новой файловой группы мы применим при внесении изменения в эту базу данных.

Выберите в этом выпадающем списке строку **<создать файловую группу>**. Появится окно создания новой файловой группы **Создание файловой группы для NewDatabase**. Введите в поле **Имя** имя файловой группы, которую так же без затей и назовем: **NewGroup** (рис. 3.16).

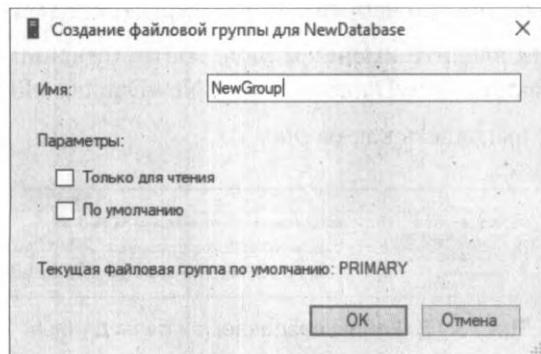


Рис. 3.16. Создание новой файловой группы

Щелкните по кнопке **ОК**. Появится новая файловая группа, которой будет принадлежать и вновь создаваемый файл **NewDatabase2**.

Установите для файла начальный размер (столбец **Начальный размер (МБ)**) в 2 Мб, введя с клавиатуры число 2 в соответствующей строке этого столбца или установив значение, используя кнопку "стрелка вверх" (для увеличения значения) или кнопку "стрелка вниз" (для уменьшения значения), как показано на рис. 3.17.



Рис. 3.17. Задание начального размера файла

Для задания характеристик роста размера файла щелкните мышью по кнопке с многоточием в правой части строки столбца **Автоувеличение**. Появится диалоговое окно **Изменить авторасширение**. В этом окне установите следующие характеристики:

- ♦ В группе радиокнопок **Увеличение размера файлов** выберите пункт **В мегабайтах**, чтобы приращение указывалось в мегабайтах, и задайте величину приращения 2, введя это значение вручную или щелкая мышью кнопки "стрелка вверх" или "стрелка вниз" в правой части окна ввода значения.
- ♦ В группе радиокнопок **Максимальный размер файла** выберите пункт **Ограниченное (МБ)** (ограничение на рост размера файла). Укажите максимальный размер файла в 20 Мб, введя значение вручную или с помощью кнопок "стрелка вверх" и "стрелка вниз".

Выбранные значения показаны на рис. 3.18.

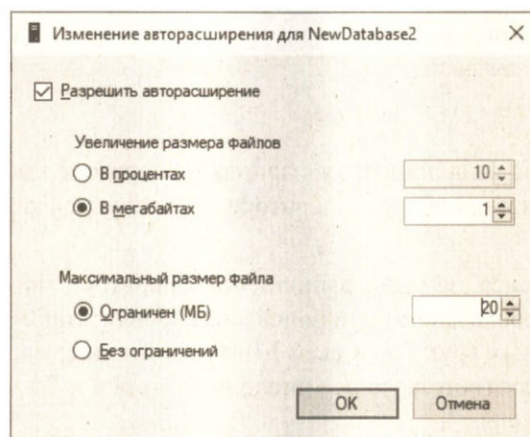


Рис. 3.18. Изменение характеристик увеличения размера файла

Щелкните мышью по кнопке **ОК**. Будут сформированы все заданные значения для второго файла данных. Список файлов создаваемой БД теперь будет выглядеть, как показано на рис. 3.19.

Файлы базы данных:							
Логическое имя	Тип файла	Файловая группа	Н...	Автоувеличение/максимальный размер	Путь	Имя файла	
NewDatabase_dat	Данные СТРОК	PRIMARY	8	С шагом по 64 МБ, без ограничений	D:\BestDatabase	NewDatabase.mdf	
NewDatabase_log	ЖУРНАЛ	Неприменимо	8	С шагом по 64 МБ, без ограничений	D:\BestDatabase	NewDatabase.ldf	
NewDatabase2	Данные СТРОК	NewGroup	2	С шагом по 1 МБ до 20 МБ	D:\BestDatabase	NewDatabase2.ndf	

Рис. 3.19. Новый список файлов базы данных

Для завершения создания базы данных щелкните по кнопке **ОК**.

3.5. Изменение базы данных

Изменять характеристики пользовательской БД, существующей в экземпляре сервера, можно оператором Transact-SQL `ALTER DATABASE` или с помощью диалоговых средств Management Studio. Рассмотрим оба варианта.

Следует помнить, что изменять характеристики можно у той базы данных, с которой в этот момент не соединены другие пользователи.

3.5.1. Изменение базы данных в языке Transact-SQL

Для изменения существующей пользовательской БД в языке Transact-SQL предусмотрен оператор `ALTER DATABASE`. Его синтаксис приведен в листинге 3.5.

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>  
};
```

Изменяться будет либо база данных, указанная в операторе по имени, либо текущая база данных, заданная 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 БД. Для выполнения нескольких действий нужно использовать соответствующее число 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 БД, которое 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 во вкладке **Сообщения** компонента Management Studio (в зависимости от того, каким средством вы сейчас пользовались) появится немножго странное сообщение об изменении имени:

база данных имя "NewBest" будет установлено.

Отобразите список имен и порядков сортировки баз данных (*пример 3.23*).

Пример 3.23. Отображение списка баз данных и порядков сортировки

```
USE master;
GO
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 первоначальное 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*.

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 БД таблицы (что мы выполним в одной из следующих 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 измене-

ния касаются столбцов, входящих в состав индексов, то нужно будет перестроить соответствующие индексы.

Отобразите опять список баз данных (см. пример 3.23) и убедитесь, что порядок сортировки нашей БД 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 БД 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 пяти 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, имена у которых начинаются с символов 'Cyrillic'. На момент написания данной главы это 68 строк.

В поле `Description` обычным текстом (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.

Перечислим имена свойств:

- ◆ 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 приведен в листинге 3.6.

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>  
}
```

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 БД 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>'
               | '<путь к MEMORY_OPTIMIZED_DATA>' }
  [, 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

Удалить существующий в базе файл данных или файл журнала транзакций позволяет предложение `REMOVE FILE`, в котором нужно указать логическое имя файла. Нельзя удалить единственный в БД файл данных или единственный файл журнала транзакций. Нельзя также удалить файл, содержащий данные.

Изменение характеристик файла

Переименовать или изменить характеристики существующего файла данных или файла журнала транзакций можно с помощью предложения `MODIFY FILE`. Синтаксическая конструкция "изменяемый файл" в этом предложении очень похожа на обычную спецификацию файла (*листинг 3.7*).

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>'
               | '<путь к MEMORY_OPTIMIZED_DATA>' }
  [, SIZE = <целое1> [ KB | MB | GB | TB ] ]
  [, MAXSIZE = { <целое2> [ KB | MB | GB | TB ] | UNLIMITED } ]
  [, FILEGROWTH = <целое3> [ KB | MB | GB | TB | % ] ]
  [, OFFLINE ]
)
```

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.

Параметр **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 значения 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 БД 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:

Файл "BestDatabase_dat" был изменен в системном каталоге. Данный новый путь будет использован при следующем запуске этой базы данных.

Имя нашего файла было изменено в системном каталоге, а новый путь будет задействован 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 БД*, 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;
```

Аналогично 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 БД 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, вам нужно дважды выполнить оператор `ALTER DATABASE` (не считая оператор перевода БД в неактивное состояние). Сказанное иллюстрирует 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 БД 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 изменяются начальный и максимальный размеры первого файла данных. Оператор приведен в *примере 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:

Сообщение 5020, уровень 16, состояние 1, строка 3

Невозможно удалить основные данные или файл журнала из базы данных.

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 БД 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, указанные в пути к файлам, должны уже существовать на диске.

Аналогично, при удалении какого-либо файла из БД 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 БД в состояние OFFLINE и, во-вторых, требуется физически средствами операционной системы изменить имя или переместить файл в другое место. После этого нужно перевести БД в состояние ONLINE.

3.5.1.4. Изменение файловых групп

В синтаксисе оператора ALTER DATABASE (см. *листинг 3.5*) указана конструкция "характеристики файловых групп". Несколько упрощенный синтаксис этой конструкции приведен в *листинге 3.8*.

Листинг 3.8. Синтаксис изменения файловых групп

```
<характеристики файловых групп> ::=  
{  
    ADD FILEGROUP <имя файловой группы>  
        [ CONTAINS FILESTREAM | CONTAINS MEMORY_OPTIMIZED_DATA ]  
    | REMOVE FILEGROUP <имя файловой группы>  
    | MODIFY FILEGROUP <имя файловой группы>  
    {  
        { READ_ONLY | READ_WRITE }  
        | DEFAULT  
        | NAME = <имя новой файловой группы>  
    }  
}
```

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 есть возможность перевести ее в состояние только для чтения (READ_ONLY) или в состояние чтения и записи (READ_WRITE), можно сделать ее группой по умолчанию (DEFAULT) и 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 в панели **Обозреватель объектов** щелкните правой кнопкой мыши по имени базы данных MultyGroup и в контекстном меню выберите элемент **Обновить**, чтобы отображались результаты изменения базы данных. Затем в том же контекстном меню выберите **Свойства**. В появившемся окне в левой верхней части щелкните по строке **Файловые группы**. В основной части окна можно увидеть, что изменения выполнены (рис. 3.20).

Имя	Файлы	Только для чтения	По умолчанию	Автоматическое увеличение всех файлов
PRIMARY	2		<input checked="" type="checkbox"/>	<input type="checkbox"/>
MultyGroup0	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
MultyGroup2	0	<input type="checkbox"/>	<input type="checkbox"/>	

Рис. 3.20. Измененный список файловых групп

В окне видно, что новая файловая группа MultyGroup2 не содержит файлов. Чтобы вновь созданную файловую группу добавить один или более файлов, нужно соответствующее число раз выполнить оператор ALTER DATABASE.

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 в предложении SET оператора 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.
- ◆ **{ ONLINE | OFFLINE | EMERGENCY }** — состояние базы данных.
- ◆ **CONTAINMENT = { NONE | PARTIAL }** — вид базы данных (обычная, частичная автономная).
- ◆ **{ READ_ONLY | READ_WRITE }** — возможность изменения данных БД.
- ◆ **{ SINGLE_USER | RESTRICTED_USER | MULTI_USER }** — допустимое число пользователей, подключаемых к БД.
- ◆ **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 }` — задает 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. Со всем не обязательно для реальной БД, 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

С помощью диалоговых средств 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 БД и в контекстном меню выбрать элемент **Переименовать**. Имя базы данных станет доступным для изменения непосредственно в панели **Обозреватель объектов**. Такой же эффект можно получить, если аккуратно щелкнуть мышью по самому имени базы данных в **Обозревателе объектов**.

Чтобы отобразить окно свойств базы данных, нужно в панели **Обозреватель объектов** щелкнуть правой кнопкой мыши по имени БД и в контекстном меню выбрать элемент **Свойства**. Появится окно свойств.

3.5.2.2. Изменение файлов базы данных

Если в окне свойств базы данных BestDatabase выбрать вкладку **Файлы**, то появится список файлов этой БД, как показано на *рис. 3.21*.

Файлы базы данных:						
Логическое имя	Тип файла	Файловая группа	Раз...	Автоматическое/максимальный...	Путь	Имя файла
BestDatabase_dat	Данные СТРОК	PRIMARY	100	С шагом по 1 МБ до 1000 МБ	D:\BestDatabase	WinnerNew.ndf
BestDatabase_log	ЖУРНАЛ	Неприменимо	2	С шагом по 1 МБ до 30 МБ	D:\BestDatabase	WinnerNew.ldf

Рис. 3.21. Характеристики файлов базы данных BestDatabase

В этой вкладке мы можем не только изменять характеристики существующих файлов, но также добавлять и удалять файлы.

Здесь можно изменить логическое имя файла, начальный размер файла и характеристики увеличения размера. 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 БД невозможен просмотр ее свойств.

Сейчас удалить ни один из существующих файлов базы BestDatabase мы не можем, потому что она содержит минимальное число требуемых файлов — один файл данных и один файл журнала транзакций. Поэтому кнопка **Удалить** неактивна.

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>. Следует заметить, 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. В правой части строки появится управляющий элемент, позволяющий изменять размер на 1 Мб в сторону увеличения или уменьшения (рис. 3.22). Размер также можно менять, вводя с клавиатуры в этом поле нужное значение.

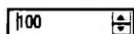


Рис. 3.22. Изменение начального размера файла

После внесения изменения следует перевести фокус ввода на любой другой элемент.

Для изменения характеристики увеличения размера (столбец **Автоувеличение**) 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. 3.23 показано такое диалоговое окно для файла журнала транзакций базы данных BestDatabase.

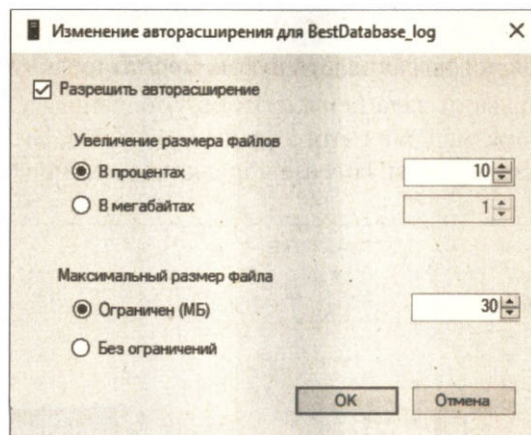


Рис. 3.23. Изменение характеристик увеличения размера файла

Здесь отмечен флажок допустимости автоматического увеличения размера (**Разрешить авторасширение**). Если увеличение размера возможно, то в группе радиокнопок **Увеличение размера файлов** можно указать, в каких единицах размер файла должен увеличиваться: в процентах или в мегабайтах.

Можно задать максимальный размер файла в группе радиокнопок **Максимальный размер файла**: максимальный (ограниченный) размер файла в мегабайтах или неограниченный размер файла.

Для добавления нового файла (данных или журнала транзакций) нужно во вкладке **Файлы** щелкнуть по кнопке **Добавить**. В список файлов будет добавлена новая строка, где отдельные столбцы содержат пустые значения, а для других установлены значения по умолчанию (рис. 3.24).

Файлы базы данных:						
Логическое имя	Тип файла	Файловая группа	Раз...	Автоматическое увеличение/максимальный размер	Путь	Имя файла
BestDatabase_dat	Данные ...	PRIMARY	100	С шагом по 1 МБ до 1000 МБ	D:\BestData...	WinnerNew.mdf
BestDatabase_log	ЖУРНАЛ	Неприменимо	2	С шагом по 1 МБ до 30 МБ	D:\BestData...	WinnerNew.ldf
	Данные ...	PRIMARY	8	С шагом по 64 МБ, без ограничений	C:\Program F...	

Рис. 3.24. Добавление нового файла в базу данных BestDatabase

Вначале добавим файл данных. Зададим ему логическое имя. 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 **Тип файла** мы не меняем, а оставляем значение, установленное по умолчанию, — **Данные СТРОК**. Значение поля **Файловая группа**, заданное как **PRIMARY**, не изменяем. Начальный размер файла, установленный в 8 Мб по умолчанию, также оставим без изменения. Не станем менять и величину автоматического увеличения размера файла. 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 **Путь** и с клавиатуры набрать полный путь к файлу, не забыв указать имя внешнего устройства. Напомню, что все каталоги в этом пути уже должны существовать на выбранном вами носителе.

При использовании диалоговых средств нужно просто щелкнуть мышью по кнопке с многоточием [...] справа от заданного пути по умолчанию в поле **Путь**. Появится окно выбора пути для размещения этого файла (рис. 3.25). Выберите диск D: и уже существующий наш каталог BestDatabase и щелкните по кнопке **ОК**.



Рис. 3.25. Выбор папки для размещения нового файла базы данных

В поле имени файла введем имя создаваемого файла **Winner2.ndf**.

Второй файл данных для нашей базы данных BestDatabase мы создали. Теперь создадим второй файл журнала транзакций.

Щелкните в окне по кнопке **Добавить**. В списке файлов появится новая строка. Задайте логическое имя файла **B2_log**. Щелкните по полю типа файла и из выпадающего списка выберите значение **ЖУРНАЛ**.

Выберите каталог для размещения файла D:\BestDatabase. Имя файла введите Winner2.ldf.

Теперь новый список файлов этой базы данных будет выглядеть следующим образом (рис. 3.26).

Файлы базы данных:						
Логическое имя	Тип файла	Файловая группа	Раз...	Автоматическое увеличение/максимальный размер	Путь	Имя файла
BestDatabase_dat	Данные ...	PRIMARY	100	С шагом по 1 МБ до 1000 МБ	D:\BestData...	WinnerNew.mdf
BestDatabase_log	ЖУРНАЛ	Неприменимо	2	С шагом по 1 МБ до 30 МБ	D:\BestData...	WinnerNew.ldf
B2_dat	Данные ...	PRIMARY	8	С шагом по 64 МБ, без ограничений	D:\BestData...	Winner2.ndf
B2_log	ЖУРНАЛ	Неприменимо	8	С шагом по 64 МБ, без ограничений	D:\BestData...	Winner2.ldf

Рис. 3.26. Добавленные два файла в базу данных BestDatabase

Чтобы созданные на предыдущих шагах добавления были фактически выполнены для текущей базы данных, щелкните по кнопке **ОК**. 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 **ОК** вы получите следующее сообщение (рис. 3.27).

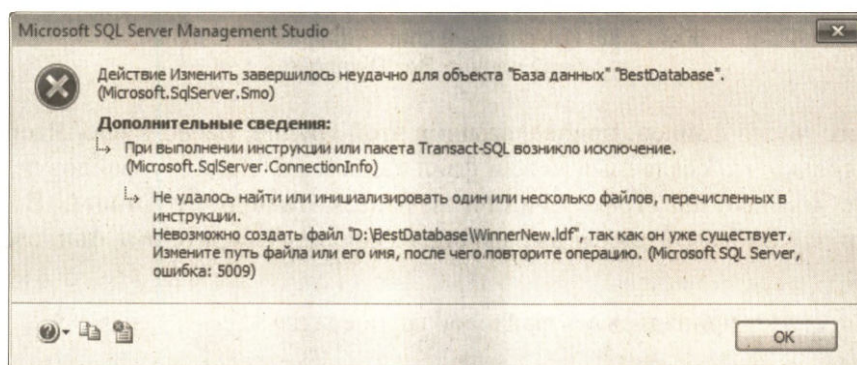


Рис. 3.27. Сообщение о дублировании имен файлов

Для удаления любого файла базы данных (кроме первичного) нужно во вкладке **Файлы** щелкнуть мышью по кнопке **Удалить** 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 **Отмена**.

Если вы выделяете на форме файл, который не может быть удален, то кнопка **Удалить** будет в недоступном состоянии.

3.5.2.3. Изменение файловых групп базы данных

Для изменения файловых групп базы BestDatabase щелкните мышью по строке **Файловые группы** в панели **Выбор страницы**. Появится список файловых групп (рис. 3.28).

Имя	Файлы	Только для чтения	По умолчанию
PRIMARY	2		<input checked="" type="checkbox"/>

Рис. 3.28. 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 **Добавить файловую группу** в нижней части этого окна. В списке появится новая строка. В поле имени введите имя файловой группы, например SECOND (рис. 3.29).

Имя	Файлы	Только для чтения	По умолчанию
PRIMARY	2		<input checked="" type="checkbox"/>
SECOND	0	<input type="checkbox"/>	<input type="checkbox"/>

Рис. 3.29. Измененный список файловых групп базы данных BestDatabase

Видно, что число файлов, принадлежащих этой группе, равно нулю. Чтобы переместить только что созданный новый файл данных B2_dat в эту файловую группу, щелкните мышью по строке **Файлы** в панели **Выбор страницы**. В столбце **Filegroup** для файла B2_dat из выпадающего 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 **ОК**.

3.5.2.4. Изменение других характеристик базы данных

Общие характеристики базы данных можно изменять, если в окне просмотра свойств этой БД в панели **Выбор страницы** выбрать вкладку **Параметры** (рис. 3.30).

Во-первых, здесь можно изменить набор символов для базы данных, выбрав в выпадающем списке **Параметры сортировки** нужный набор. Заменяем установленный нами по умолчанию при создании БД 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 **Параметры сортировки**. Появится выпадающий список доступных порядков сортировки. Выберите из списка `French_CI_AI`.

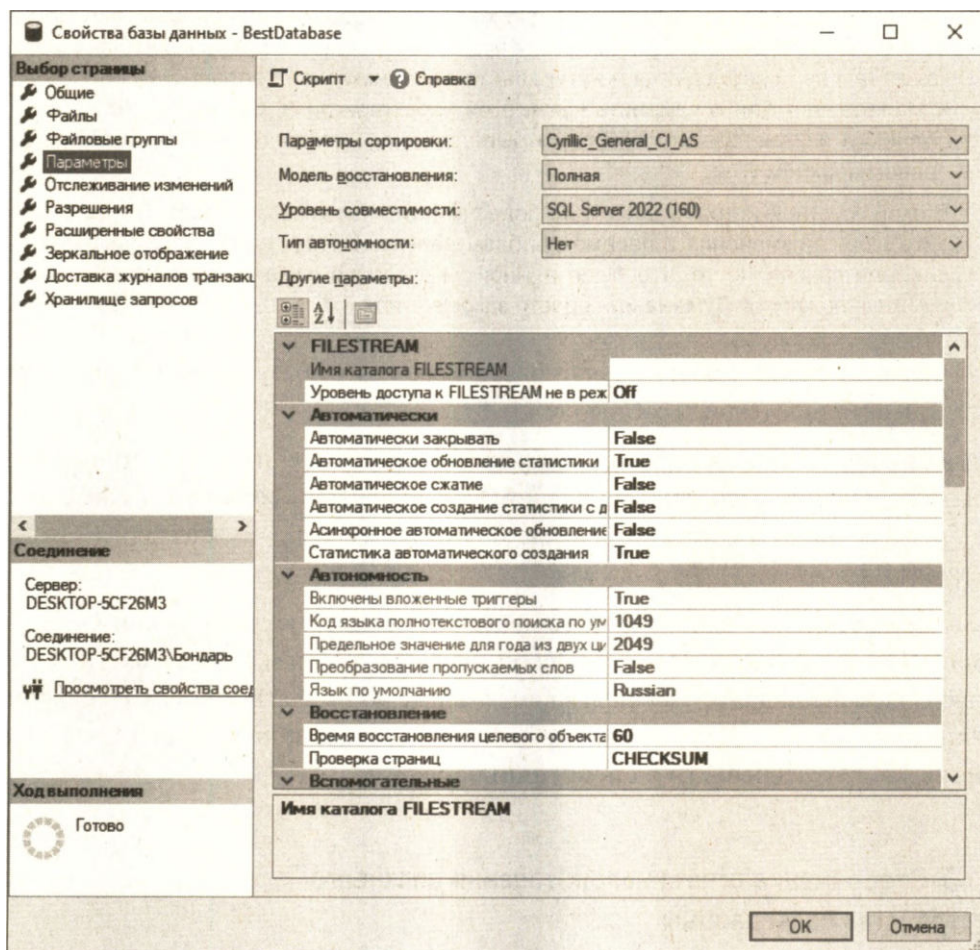


Рис. 3.30. Вкладка свойств базы данных

Существует понятие уровня совместимости с предыдущими версиями. Для вновь создаваемых систем обработки данных следует оставить для уровня совместимости значение по умолчанию.

В основном списке режимов 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.

Чтобы изменить значение характеристики, нужно щелкнуть мышью по этой строке; в правой части значения характеристики появится стрелка, позволяющая вызвать выпадающий список. Чаще всего в списке присутствует только два значения — истина и ложь (**True** и **False**).

ВНИМАНИЕ!

Еще раз напомним. Чтобы проделанная вами работа по изменению любых характеристик БД, ее файлов и файловых групп действительно была зафиксирована в базе данных, необходимо в окне просмотра свойств БД в завершение всех действий щелкнуть по кнопке **ОК**. 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 его правом верхнем углу.

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 **Отмена** 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 БД Database Engine база находится в состоянии ONLINE. Для этих целей БД нужно перевести в состояние OFFLINE.

Чтобы перевести базу данных в неактивное состояние, нужно в панели **Обозреватель объектов** щелкнуть правой кнопкой мыши по имени БД, в контекстном меню выбрать элемент **Задачи** и в открывшемся подменю выбрать элемент **Перевести в автономный режим**. Для перевода в активное состояние в этом подменю нужно выбрать элемент **Перевести в оперативный режим**.

3.5.2.5. Отображение отчета использования дискового пространства базы данных

Здесь хочу сказать два слова о тех удобствах, 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. В панели **Обозреватель объектов** щелкните правой кнопкой мыши по имени БД, в контекстном меню выберите элемент **Отчеты**, затем элементы **Стандартный отчет** и **Занято место на диске**. В результате будет создан наглядный отчет, отображающий использование базой данных внешнего пространства (рис. 3.31).

В процессе заполнения базы данных необходимыми данными можно периодически создавать такой отчет, который будет отображать текущее положение с использованием дискового пространства.

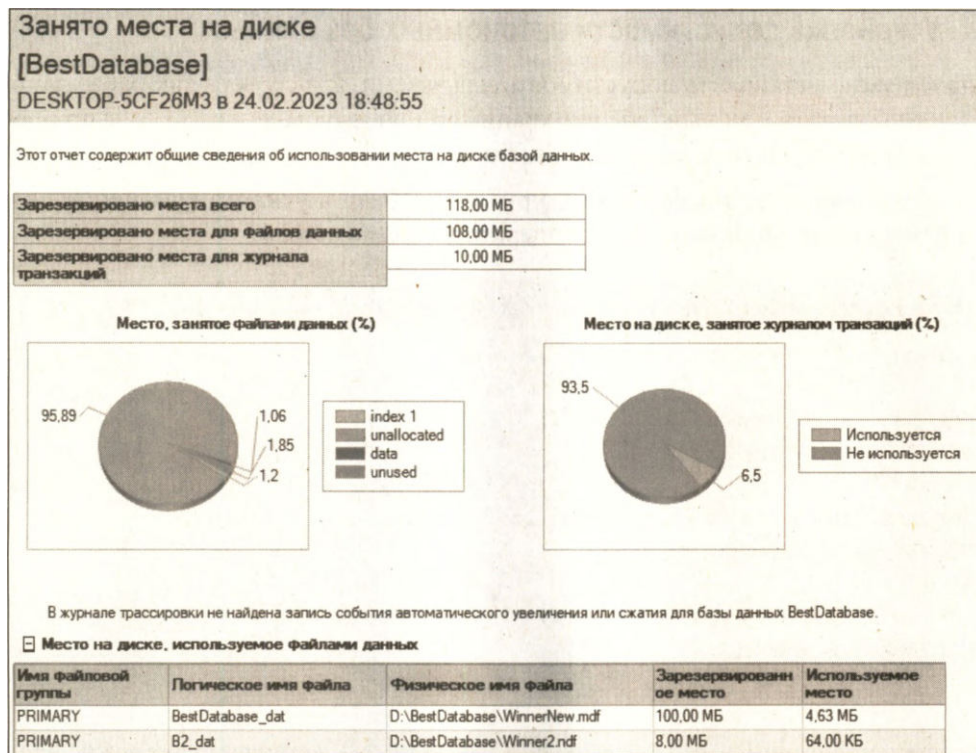


Рис. 3.31. Отчет использования дискового пространства файлами базы данных

3.5.3. Удаление базы данных диалоговыми средствами Management Studio

Для удаления базы данных в Management Studio нужно в **Обозревателе объектов** щелкнуть правой кнопкой мыши по имени БД и в контекстном меню выбрать элемент **Удалить**.

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. Поэтому их очень просто перемещать между различными экземплярами сервера БД, переносить на другие компьютеры.

Созданную "обычную" базу данных можно также перевести в автономную.

Рассмотрим вкратце все действия, необходимые для создания автономной БД 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 помощи 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
```

Здесь значение опции `contained database authentication` устанавливается в 1, что позволяет создавать автономные базы данных. Команда `RECONFIGURE` нужна, чтобы изменения вступили в силу.

В результате выполнения скрипта будет получено сообщение:

Параметр конфигурации "show advanced options" изменен с 0 на 1. Выполните инструкцию `RECONFIGURE` для установки.

Параметр конфигурации "contained database authentication" изменен с 0 на 1. Выполните инструкцию `RECONFIGURE` для установки.

Параметр конфигурации "show advanced options" изменен с 1 на 0. Выполните инструкцию `RECONFIGURE` для установки.

При использовании диалоговых средств Management Studio в окне **Обозреватель объектов** щелкните правой кнопкой мыши по имени сервера БД и в контекстном меню выберите элемент **Свойства**. В появившемся окне **Свойства сервера** в левой части выберите вкладку **Дополнительно** (рис. 3.32).

В основной части окна из выпадающего списка поля **Разрешить автономные базы данных** выберите пункт **True**. Закройте окно свойств сервера, щелкнув по кнопке **ОК**.

Чтобы изменения вступили в силу, нужно перезапустить сервер. В окне **Обозреватель объектов** щелкните правой кнопкой мыши по имени сервера БД и в контекстном меню выберите элемент **Перезапустить**. В появившемся окне подтверждения перезапуска сервера щелкните по кнопке **Да**. Через некоторое время сервер будет заново запущен, изменения будут приняты.

В версии 2022 не требуется перезапускать сервер.

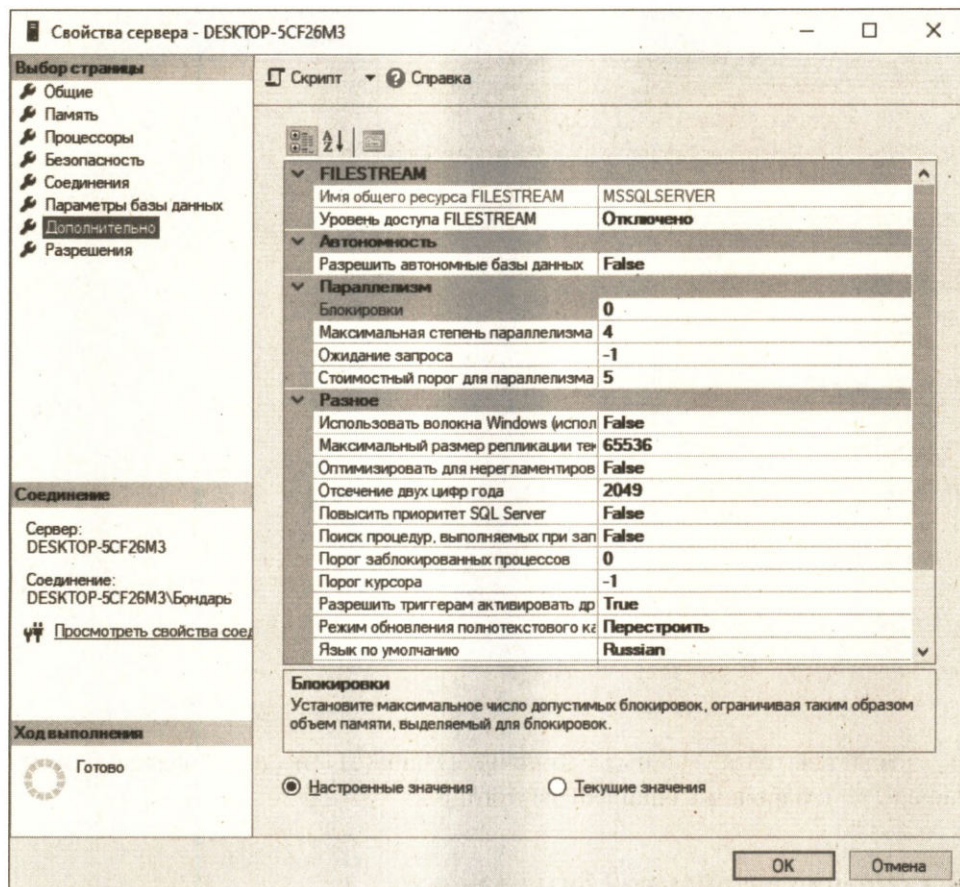


Рис. 3.32. Вкладка Дополнительно свойств сервера

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, предварительно создав на диске D: каталог ContainedDatabase (*пример 3.37*).

Пример 3.37. Создание автономной БД 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 БД ContainedUser с паролем ContainedPassword.

3.6.3. Создание автономной базы данных диалоговыми средствами Management Studio

Для создания автономной базы данных в окне **Обозреватель объектов** щелкните правой кнопкой мыши по папке **Базы данных** и в контекстном меню выберите элемент **Создать базу данных**. В окне **Создание базы данных** в поле **Имя базы данных** введите имя базы данных: **ContainedDatabase2**. В поле **Путь** введите один и тот же путь к обоим файлам БД (файлу данных и файлу журнала транзакций): **D:\ContainedDatabase**. В поле **Имя файла** укажите имена файлов базы данных (рис. 3.33).

Файлы базы данных:						
Логическое имя	Тип файла	Файловая группа	Нач...	Автоувеличение/максим...	Путь	Имя файла
ContainedDatabase2	Данные ...	PRIMARY	8	С шагом по 64 МБ, б...	D:\ContainedDatabase ...	ContainedDatabase2.mdf
ContainedDatabase2_log	ЖУРНАЛ	Неприменимо	8	С шагом по 64 МБ, б...	D:\ContainedDatabase ...	ContainedDatabase2.ldf

Рис. 3.33. Создаваемая база данных contained

Во вкладке **Параметры** в поле **Тип автономности** выберите из выпадающего списка значение **Частично**. Для завершения создания базы данных щелкните по кнопке **ОК**.

ВНИМАНИЕ!

Чтобы все действия по созданию автономной базы данных и ее пользователя выполнялись правильно, вам следует запустить Management Studio с правами (от имени) администратора.

3.6.4. Создание автономного пользователя в Management Studio

Чтобы создать в базе данных ContainedDatabase2 нового автономного пользователя с паролем, в **Обозревателе объектов** раскройте узел **Базы данных**, затем **ContainedDatabase2** и **Безопасность**. Щелкните правой кнопкой мыши по строке **Пользователи** и выберите в контекстном меню строку **Создать пользователя**. Появится окно **Пользователь базы данных - Создать**. Из выпадающего списка **Тип пользователя** выберите элемент **Пользователь SQL с паролем**. В поле **Имя пользователя** введите имя пользователя: **ContainedUser**, в поле **Пароль** и в поле **Подтверждение пароля** введите: **ContainedPassword** (рис. 3.34).

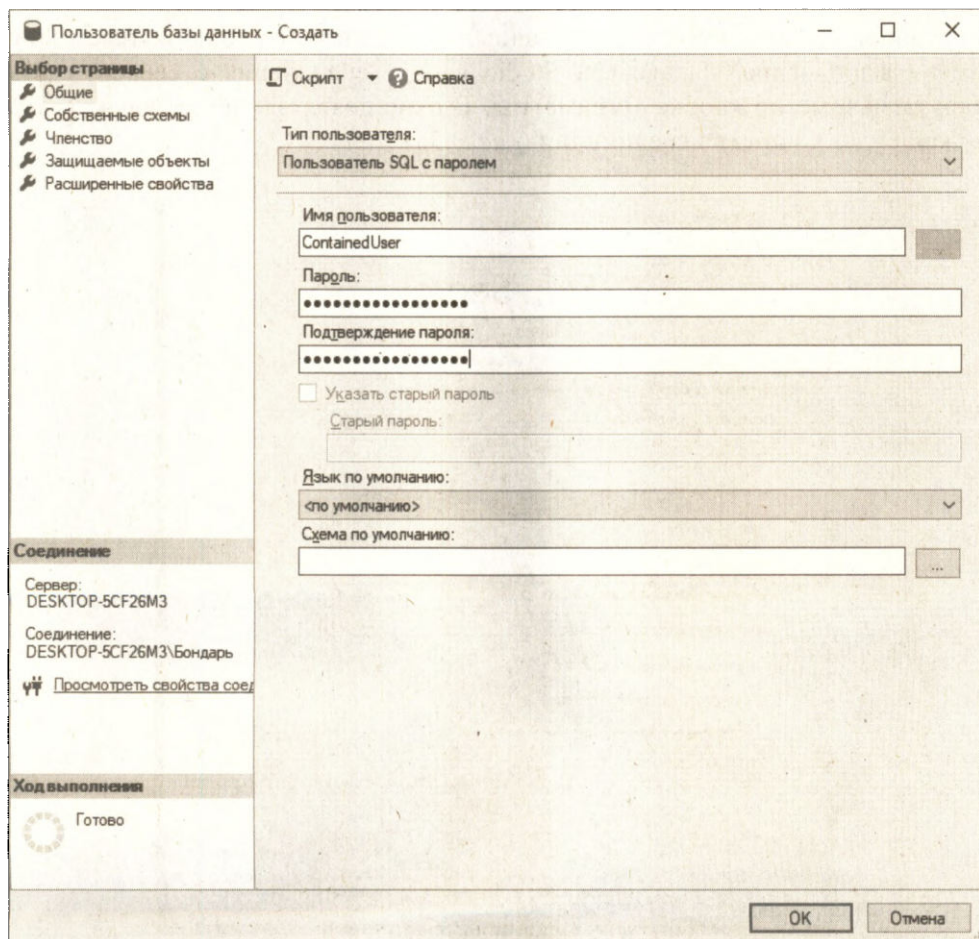


Рис. 3.34. Задание пользователя базы данных. Вкладка **Общие**

В левой части окна выберите вкладку **Членство**. Установите флажок в поле **db_owner** (рис. 3.35). Щелкните по кнопке **ОК**.

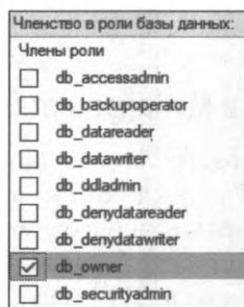


Рис. 3.35. Задание пользователя базы данных. Вкладка **Членство**

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 **Параметры**. В окне появятся три вкладки. Текущей вкладкой будет **Свойства соединения** (рис. 3.36).

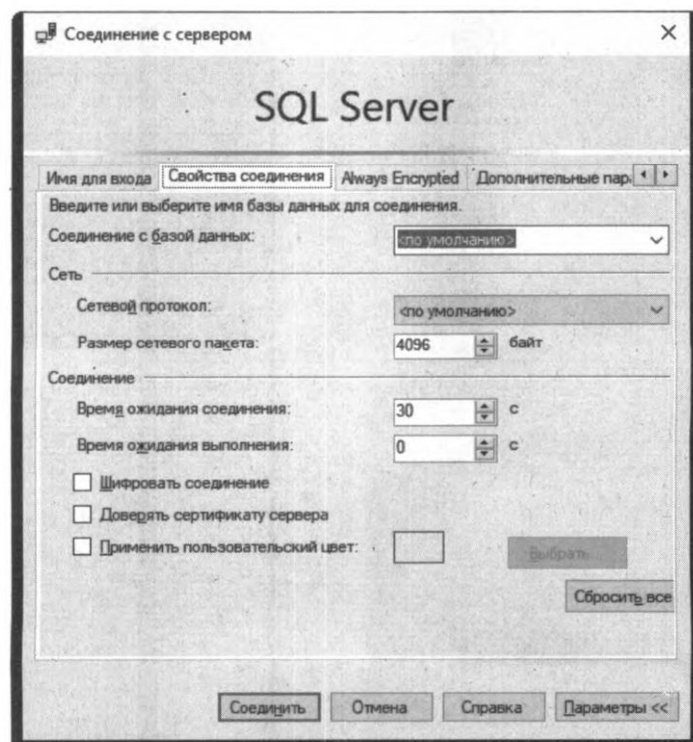


Рис. 3.36. Соединение с базой данных. Вкладка **Свойства соединения**

В поле **Соединение с базой данных** нужно из выпадающего списка выбрать базу данных **ContainedDatabase1** или набрать с клавиатуры это имя.

Выберите вкладку **Имя для входа**. Из выпадающего списка **Проверка подлинности** выберите **Проверка подлинности SQL Server**, в поле **Имя входа** введите имя пользователя **ContainedUser**, в поле **Пароль** — пароль **ContainedPassword** (рис. 3.37).

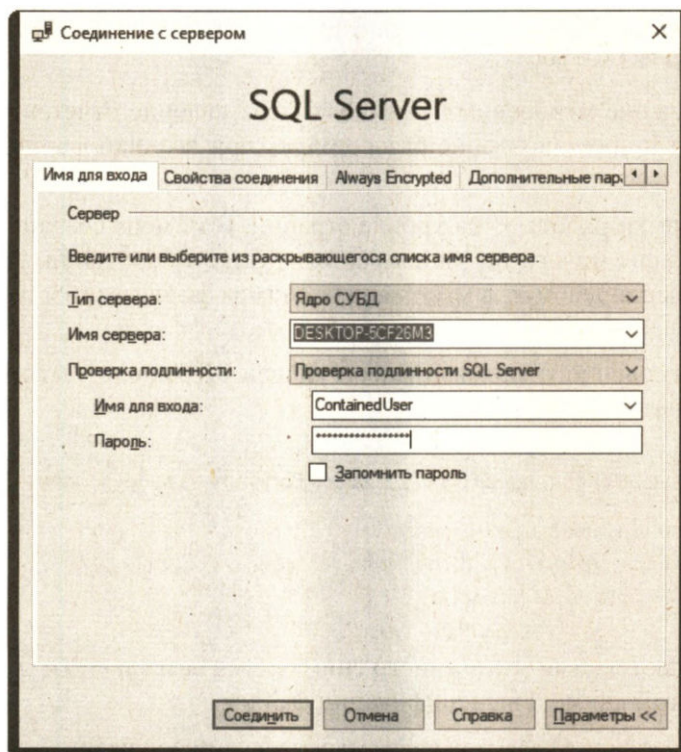


Рис. 3.37. Соединение с базой данных. Вкладка **Имя для входа**

Щелкните мышью по кнопке **Соединить**. Произойдет соединение с этой базой данных. В списке БД в **Обозревателе объектов** помимо системных баз данных будет видна только **ContainedDatabase1**.

ЗАМЕЧАНИЕ

Прежде чем щелкнуть по кнопке **Соединить**, нужно вернуться на вкладку **Свойства соединения** и повторно указать базу данных, с которой производится соединение. Иначе возникают ошибки соединения. Такое же поведение было и в SQL Server 2014.

3.7. 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 БД (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.

ВНИМАНИЕ!

Мгновенные снимки не поддерживаются в версии Express Edition. Действия по созданию мгновенного снимка, описанные в данном разделе, можно выполнить лишь на других версиях SQL Server.

Основное назначение мгновенных снимков — составление отчетов на конкретный момент времени и восстановление базы данных при возникновении 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.

Для создания мгновенного снимка существующей БД используется вариант оператора CREATE DATABASE (*листинг 3.10*).

Листинг 3.10. Синтаксис оператора CREATE DATABASE. Вариант создания мгновенного снимка БД

```
CREATE DATABASE <имя мгновенного снимка>  
    ON (<спецификация файла>) [, (<спецификация файла>)]...  
    AS SNAPSHOT OF <имя базы данных>;
```

В операторе задаются: 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 показано в *примере 3.38*.

Пример 3.38. Создание мгновенного снимка для базы данных BestDatabase

```
USE master;  
GO  
CREATE DATABASE SnapshotForBestDatabase  
ON (NAME = BestDatabase_dat,  
    FILENAME = 'D:\BestDatabase\Snapshot.mdf'),  
    (NAME = B2_dat,  
    FILENAME = 'D:\BestDatabase\Snapshot2.mdf')  
AS SNAPSHOT OF BestDatabase;  
GO
```

В результате выполнения этого скрипта в каталоге BestDatabase будут созданы файлы мгновенного снимка `Snapshot.mdf` и `Snapshot2.mdf`.

3.8. Схемы базы данных

Физически база данных представлена в одном или более файлах данных. Логически база состоит из произвольного числа схем (schema). В схемах хранятся объекты БД, 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. Объекты с одинаковыми именами могут присутствовать в различных схемах одной и той же БД.

Основное назначение схемы — повышение уровня безопасности данных для того, чтобы предоставить права на использование объектов БД только указанным группам пользователей. Таких пользователей называют принципами (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.

3.8.1. 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 перемещение объектов из одной схемы в другую. Удаляется схема оператором `DROP SCHEMA`. Удалить можно только схему, которая не содержит объектов.

3.8.1.1. Создание схемы

Синтаксис оператора создания схемы `CREATE SCHEMA` приведен в листинге 3.11.

Листинг 3.11. 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`.

3.8.1.2. Изменение схемы

Перемещение объектов между схемами одной базы данных осуществляется с помощью оператора `ALTER SCHEMA`. Синтаксис приведен в *листинге 3.12*.

Листинг 3.12. 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.

3.8.1.3. Удаление схемы

Удаление пустой схемы выполняется оператором `DROP SCHEMA` (*листинг 3.13*).

Листинг 3.13. Синтаксис оператора `DROP SCHEMA`

```
DROP SCHEMA <имя схемы>;
```

3.8.1.4. Пример создания схем

Создайте в базе данных `BestDatabase` две схемы, выполнив операторы *примера 3.39*.

Пример 3.39. 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 3.40).

Пример 3.40. 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
```

Результаты 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

Теперь удалите созданные две схемы (пример 3.41).

Пример 3.41. 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.

3.8.2. Работа со схемами в Management Studio

3.8.2.1. Отображение схем базы данных

Чтобы в Management Studio просмотреть список схем базы данных BestDatabase, нужно в **Обозревателе объектов** раскрыть список **Базы данных**, раскрыть базу данных BestDatabase, папку **Безопасность** и папку **Схемы**. Появится список схем базы данных.

3.8.2.2. Создание схемы

Чтобы создать новую схему, нужно в окне **Обозревателя объектов** щелкнуть правой кнопкой мыши по элементу **Схемы** в базе данных и в контекстном меню выбрать строку **Создать схему**.

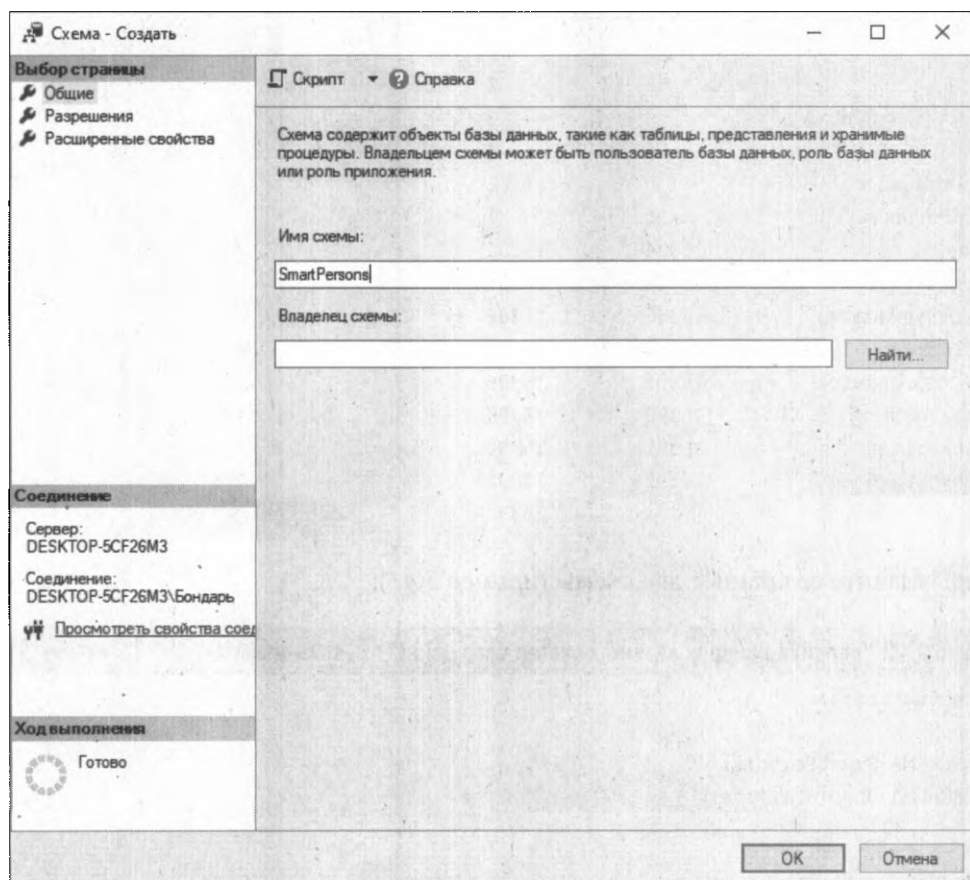


Рис. 3.38. Создание новой схемы в базе данных BestDatabase

Появится окно создания новой схемы (рис. 3.38). В поле **Имя схемы** введите имя создаваемой схемы: SmartPersons.

Имя владельца схемы можно не вводить. Туда будет подставлено имя пользователя по умолчанию — dbo. Щелкните по кнопке **ОК**.

Аналогичным образом создайте вторую схему StupidPersons.

Чтобы увидеть вновь созданные схемы в **Обозревателе объектов**, нужно щелкнуть правой кнопкой мыши по элементу **Схемы** и в контекстном меню выбрать строку **Обновить**.

3.8.2.3. Изменение существующей схемы

Диалоговыми средствами Management Studio можно изменить владельца существующей схемы. Для этого в **Обозревателе объектов** дважды щелкните мышью по схеме или щелкните правой кнопкой мыши по имени схемы и в контекстном меню выберите элемент **Свойства**. Появится окно, похожее на окно создания новой схемы.

Чтобы изменить владельца схемы, в поле **Владелец схемы** щелкните по кнопке **Найти**. Появится диалоговое окно выбора владельца схемы — пользователя или роли (рис. 3.39).

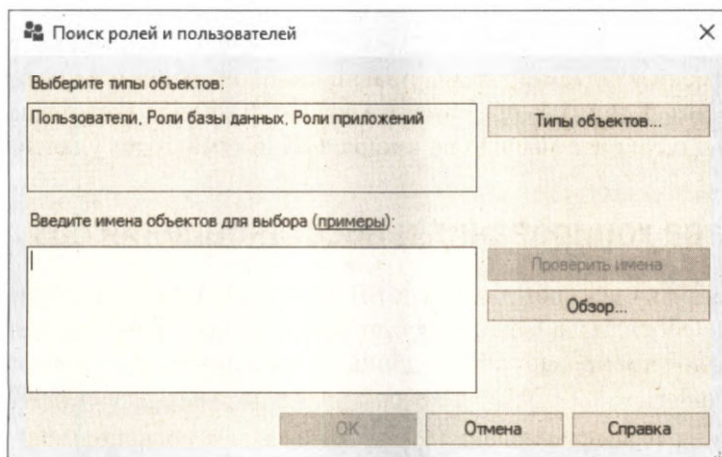


Рис. 3.39. Окно выбора владельца схемы

Щелкните по кнопке **Обзор**. Появится окно просмотра объектов базы данных, которые могут быть использованы в качестве владельцев схемы. Отметьте флажок в строке **[public]** и щелкните по кнопке **ОК** (рис. 3.40).

Для схемы будет выбран новый владелец **public**. Чтобы сохранить изменения, в окне просмотра свойств схемы щелкните по кнопке **ОК**.

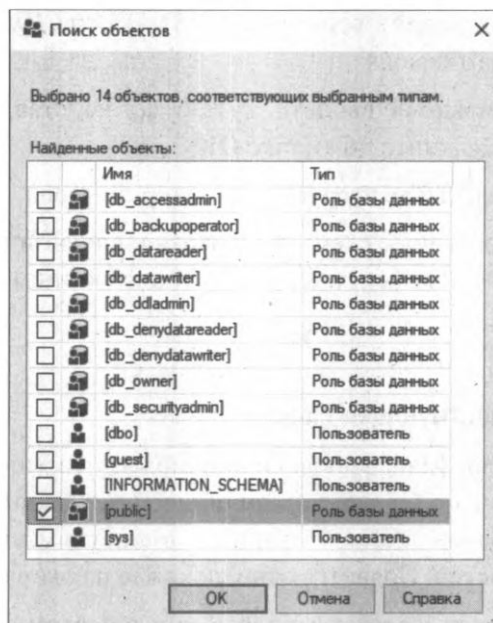


Рис. 3.40. Список возможных владельцев схемы

3.8.2.4. Удаление схемы

Удалите поочередно обе созданные схемы. Для этого в **Обозревателе объектов** щелкните по имени удаляемой схемы правой кнопкой мыши и в контекстном меню выберите элемент **Удалить** или нажмите клавишу . В окне подтверждения удаления схемы щелкните мышью по кнопке **ОК**. Схема будет удалена.

3.9. Средства копирования и восстановления баз данных

При интенсивном изменении данных в БД эту базу нужно регулярно копировать, чтобы в случае сбоев было можно восстановить данные. Рекомендуется также выполнять резервное копирование базы данных master после создания новых БД 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.

3.9.1. Использование операторов Transact-SQL для копирования/восстановления базы данных

3.9.1.1. Копирование 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 3.14*.

Листинг 3.14. Синтаксис оператора BACKUP DATABASE

```
BACKUP DATABASE <имя базы данных>  
    TO DISK = '<спецификация файла>'  
    [, DISK = '<спецификация файла>']...;
```

Этот оператор выполняет копирование всех файлов БД 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.

3.9.1.2. Восстановление базы данных

Восстановление БД невозможно, если для нее используются средства мгновенных снимков. Для удаления таких снимков базы данных BestDatabase нужно выполнить оператор:

```
DROP DATABASE SnapshotForBestDatabase;
```

Для восстановления базы данных из резервных копий предусмотрен оператор RESTORE. Его упрощенный синтаксис для полного восстановления БД приведен в *листинге 3.15*.

Листинг 3.15. Синтаксис оператора RESTORE

```
RESTORE DATABASE <имя базы данных>  
    FROM DISK = '<спецификация файла>'  
    [, DISK = '<спецификация файла>']...  
    [ WITH REPLACE ];
```

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.

Необходимо иметь в виду, что средства копирования и восстановления в SQL Server 2022 несовместимы с более ранними версиями.

3.9.2. Использование диалоговых средств Management Studio для копирования/восстановления базы данных

3.9.2.1. Копирование базы данных

Для копирования базы данных BestDatabase в Обозревателе объектов щелкните правой кнопкой мыши по имени БД, в контекстном меню выберите элементы За-

дачи | Создать резервную копию. Появится окно **Резервное копирование базы данных** (рис. 3.41).

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 **Удалить**, чтобы удалить этот вариант.

Затем щелкните по кнопке **Добавить**, чтобы задать свой диск и свой файл. Появится окно **Выбор места расположения резервной копии** (рис. 3.42).

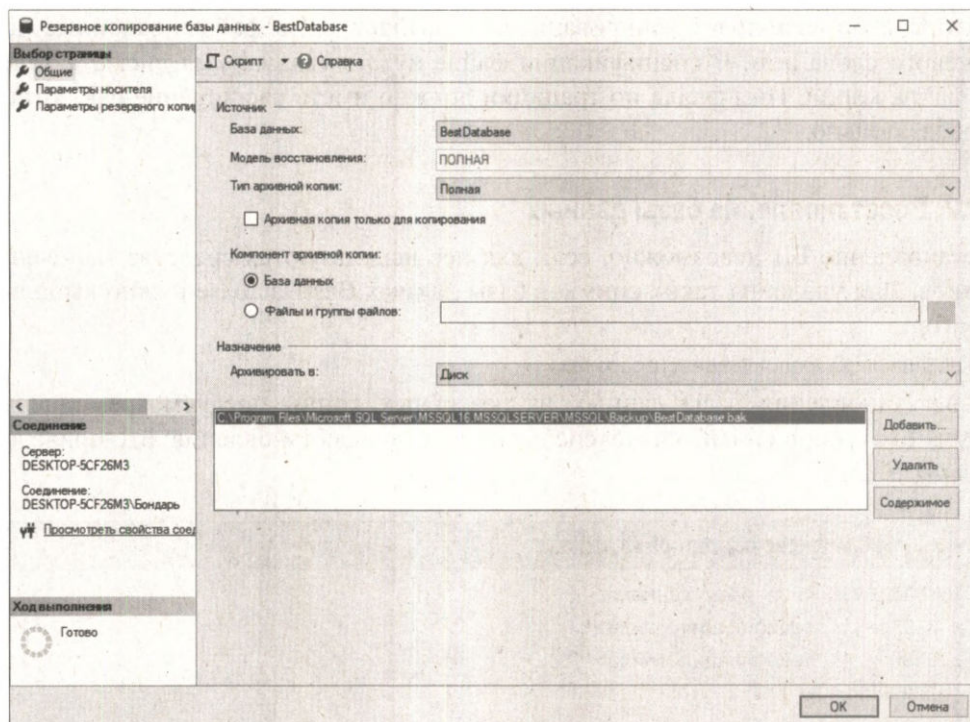


Рис. 3.41. Окно копирования базы данных

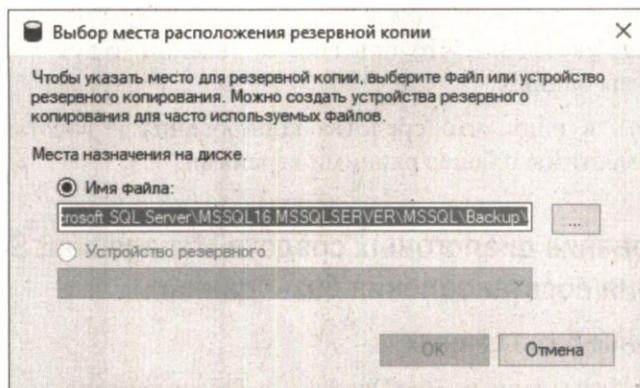


Рис. 3.42. Окно выбора файлов копии для базы данных

Здесь нужно щелкнуть по кнопке с многоточием. Появится окно **Расположение файлов базы данных**, в котором нужно задать размещение и имя файла копии. Диск и каталог выбирают из списка, как показано на *рис. 3.43*.

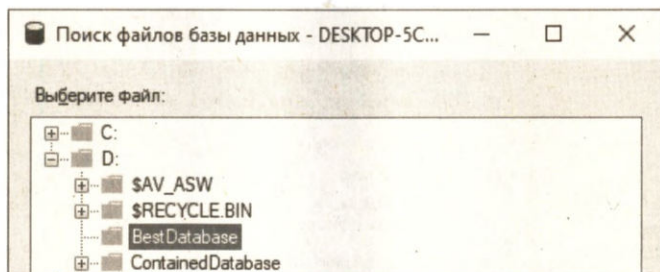


Рис. 3.43. Окно выбора размещения файлов копии

Имя файла копии (BackupBest.bak) вводят в нижней части окна (*рис. 3.44*).

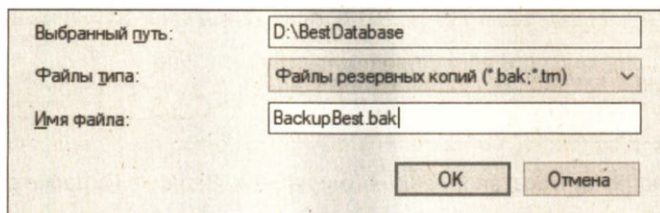


Рис. 3.44. Задание имени файла копии

После щелчка по кнопке **ОК** в этом и в предыдущем окне будет выполнено копирование и появится сообщение об успешном завершении копирования.

3.9.2.2. Восстановление базы данных

Опять напоминаю: восстановление базы данных невозможно, если для нее используются средства мгновенных снимков. Для удаления снимков БД BestDatabase нужно выполнить оператор:

```
DROP DATABASE SnapshotForBestDatabase;
```

Для восстановления базы данных BestDatabase в **Обозревателе объектов** щелкните правой кнопкой мыши по имени БД, в контекстном меню выберите элементы **Задачи | Восстановить | База данных**. Появится окно **Восстановление базы данных**. Здесь нужно перейти во вкладку **Параметры** и отметить поле **Перезаписать существующую базу данных (WITH REPLACE)** (*рис. 3.45*).

Во вкладке **Общие** нужно отметить радио-кнопку **База данных** и из выпадающего списка выбрать имя восстанавливаемой базы данных (*рис. 3.46*).

Во вкладке **Файлы** (*рис. 3.47*) можно изменить пути и имена физических файлов базы данных.

После щелчка по кнопке **ОК** будет выполнено восстановление и появится соответствующее информационное сообщение (*рис. 3.48*).

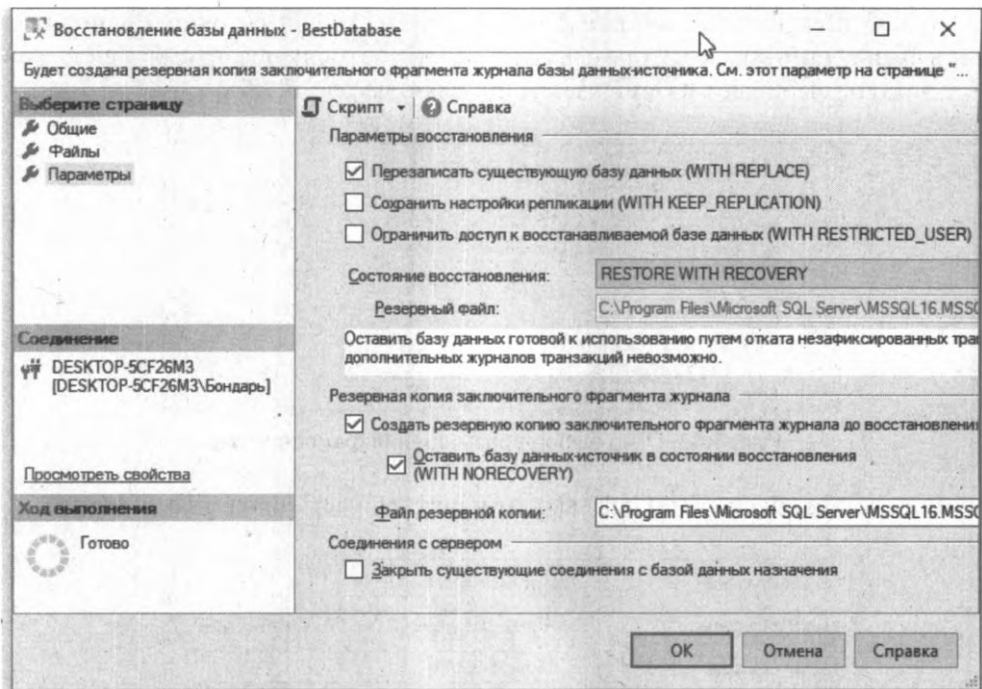


Рис. 3.45. Восстановление базы данных. Вкладка Параметры

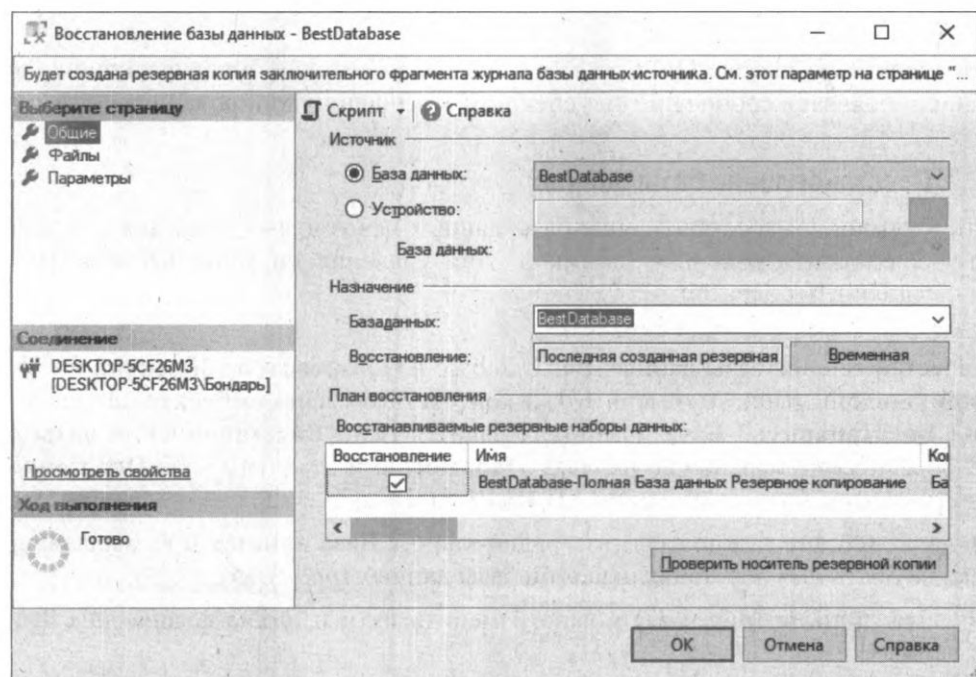


Рис. 3.46. Восстановление базы данных. Вкладка Общие

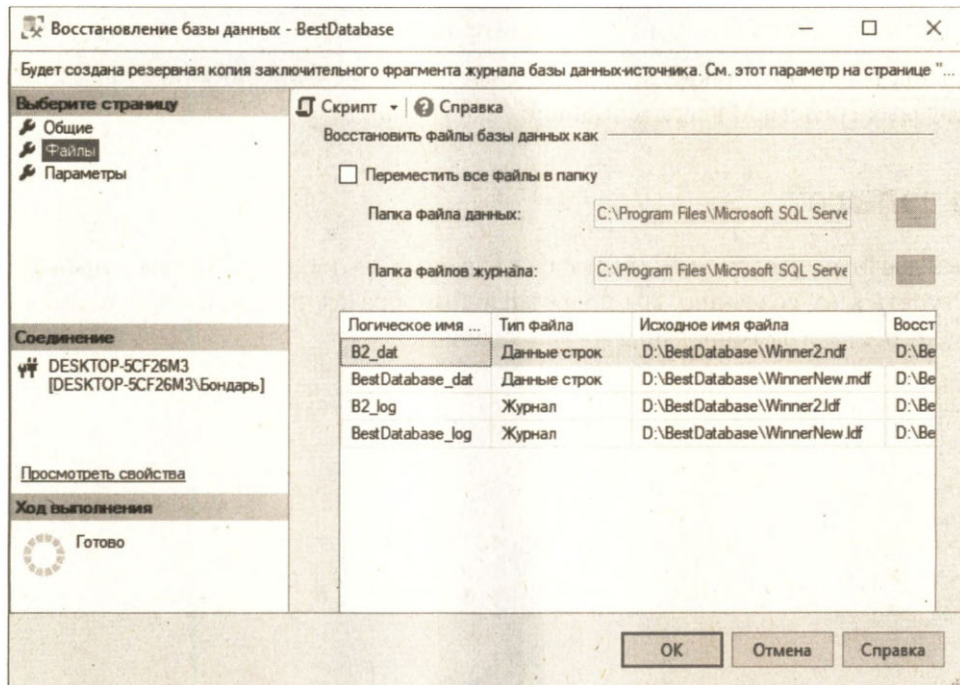


Рис. 3.47. Восстановление базы данных.
Вкладка **Файлы**

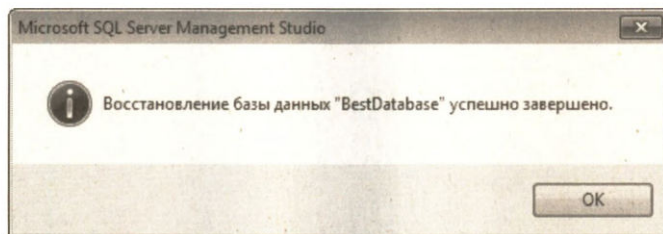


Рис. 3.48. Сообщение об успешном восстановлении базы данных

3.10. Домашнее задание

Сейчас предлагаю выполнить ряд работ по созданию базы данных, которая может вам пригодиться в реальной жизни.

Создайте БД с именем 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 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.

Что дальше?

Самый важный и сложный объект базы данных — таблицы. Однако прежде чем приступить к их созданию, мы подробнейшим образом рассмотрим фундаментальное понятие программирования — типы данных.

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 задает множество допустимых значений для элемента данных и множество допустимых операций, применимых к элементу данных. Кроме того, тип данных определяет способ внутреннего хранения соответствующего элемента, т. е. физический аспект.

В этой 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 для совместимости со стандартом SQL имеют синонимы. Эти синонимы также приводятся для соответствующих типов данных.

Материала в настоящей главе достаточно много. Если вы впервые знакомитесь с типами данных в реляционных БД, 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.
- Приблизительные 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). Типы данных:
 - ◊ CHAR,
 - ◊ VARCHAR,
 - ◊ TEXT.
- Символьные строки в Юникоде (Unicode Character Strings). Типы данных:
 - ◊ NCHAR,
 - ◊ NVARCHAR,
 - ◊ NTEXT.

◆ **Дата и время (Date and Time).** Содержит типы данных:

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

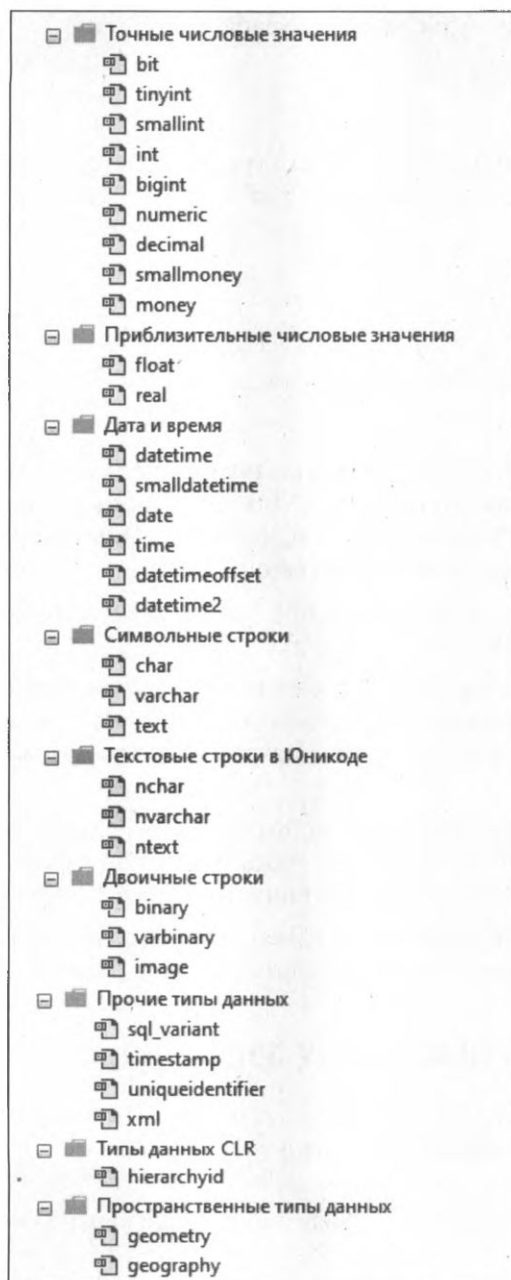


Рис. 4.1. Список типов данных, отображаемых в Management Studio

◆ **Двоичные строки (Binary Strings). Типы данных:**

- BINARY,
- VARBINARY,
- IMAGE.

◆ Пространственные типы данных (Spatial Data Types):

- GEOMETRY,
- GEOGRAPHY.

◆ Другие типы данных (Other Data Types). Сюда включены типы данных, не очень подходящие для других категорий:

- SQL_VARIANT,
- TIMESTAMP,
- UNIQUEIDENTIFIER,
- HIERARCHYID (еще называется тип данных CLR),
- XML.

Список системных типов данных можно просмотреть и в компоненте Management Studio в окне **Обозреватель объектов**. Для этого нужно раскрыть папку **Базы данных**, любую пользовательскую БД, например **BestDatabase**, далее **Программирование**, **Типы и**, наконец, **Системные типы данных**.

Основная часть списка типов данных при раскрытых подгруппах **Системные типы данных** показана на *рис. 4.1*.

Если к строке, соответствующей любому типу данных в этом окне, подвести курсор мыши, то рядом с курсором появится подсказка (hint), где помимо названия типа данных будет указан 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 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**.

Синтаксис оператора **DECLARE** для объявления локальных переменных показан в *листинге 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> ] ] ... ;
```

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 так 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).

Присвоить другое, новое значение, локальной переменной в тексте пакета можно при помощи 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 допустимы арифметические, а также логические побитовые операции 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 посредством 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

Рассмотрим следующую классификацию типов данных, традиционно входящих в SQL Server в эту группу Numeric. Выделим следующие подгруппы:

- ♦ Тип данных BIT — его определяют как целочисленный тип данных, хотя следовало бы назвать условно логическим типом.

- ◆ 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.

В три последние подгруппы входят типы данных, которые мы назовем истинно числовыми данными.

Для числовых типов данных (кроме 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 помощи функции CAST() или CONVERT(). В числовые типы данных с некоторыми ограничениями, которые мы рассмотрим чуть позже, можно преобразовывать "правильные" строки, т. е. 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 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 БД.

К выражениям, возвращающим логические значения, можно применять операции дизъюнкцию (логическое ИЛИ, OR), конъюнкцию (логическое И, AND) и отрицание (НЕ, NOT). Эти операции можно применять только в условиях соответствующих операторов.

ров. В языке Transact-SQL помимо операторов IF и CASE существуют две логические функции: IIF() и CHOOSE(). Они появились еще 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 число параметров. По-хорошему, параметр "индекс" должен принимать целочисленное значение. Однако если он представлен дробным числом, то дробная часть отбрасывается без округления. Функция возвращает то значение из списка, номер которого равен числу в индексе. Первое значение имеет номер 1. Если значение индекса превышает число элементов в списке значений, является нулем или отрицательным числом, то функция вернет NULL.

4.3.1. Тип данных BIT

Множество 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 нечувствительны к регистру. Такие преобразования дадут, соответственно, 1 и 0. Значение 1 соответствует в этом типе данных значению "истина", а 0 — "ложь".

Для проверки поведения типа данных BIT при выполнении различных операций в командной строке, в PowerShell или в SQL Server Management Studio выполните следующие 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

Надеюсь, вы помните, что функция `CAST()` преобразует константу или выражение 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 операторе осуществляется отрицание ложного значения. Последний элемент иллюстрирует дизъюнкцию, логическое или.

В табл. 4.1–4.3 приводятся таблицы истинности для трех логических операций (отрицания, дизъюнкции и конъюнкции) при использовании в качестве операндов логических значений 1 (истина) и 0 (ложь) 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 операндов

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

Таблица 4.3 (окончание)

Операнд 1	Операнд 2	Операнд 1 & Операнд 2
1	NULL	NULL
0	NULL	NULL
NULL	1	NULL
NULL	0	NULL
NULL	NULL	NULL

ЗАМЕЧАНИЕ

Должен сказать, что приведенные таблицы истинности, принятые в MS SQL Server, мне категорически не нравятся. Это не соответствует операциям нормальной трехзначной логики. Например, дизъюнкция двух операндов, где один является истинным, должна возвращать истину, даже если другой операнд NULL. Правда, такие таблицы истинности используются только для типа данных BIT, в остальных вариантах они правильные.

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 число столбцов в таблице типа данных BIT потребует уже нескольких байтов.

4.3.2. Целочисленные типы данных TINYINT, SMALLINT, INT, BIGINT

Эти типы данных позволяют хранить целые числа. Они различаются числом 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

Таблица 4.4 (окончание)

Тип данных	Размер (байт)	Диапазон значений
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 употреблять синоним `INTEGER`. Синонимы для некоторых типов данных вводятся для соответствия стандарту SQL.

Если при делении целых чисел в результате получается дробное число, то дробная часть просто отбрасывается, округление не проводится. Такое же поведение будет и при преобразовании дробного числа в целое в случае использования функции `CAST()`. При желании вы 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 округление тоже не выполняется. Например, в результате следующего преобразования

```
SELECT CAST(25.9 AS TINYINT);
```

мы получим число 25. А вот такое преобразование, как

```
SELECT CAST(259 AS TINYINT);
```

выдаст ошибку — арифметическое переполнение (*arithmetic overflow*), поскольку тип данных `TINYINT` дает возможность представлять только положительные числа (точнее, 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 одинаково: дробная часть отбрасывается без округления числа. Например, допустимо такое преобразование:

```
SELECT CAST(25.98E+1 AS INT);
```

Результатом, как и ожидалось, будет число 259.

Если дробные числа можно преобразовать к целому числу при соответствии исходного числа размеру целого, отводимого для хранения значения, то 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 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 слишком 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. Наша предыдущая попытка преобразования 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. Проще было бы сразу эту константу записать как число с плавающей точкой, 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 типу данных даст значение `NULL`.

Для внутреннего хранения констант система выбирает характеристики, минимально возможные для каждой конкретной константы. Если числовая константа неотрицательная и не превышает значение 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 (/), а также операция получения остатка от деления (%). Здесь, в отличие от целых чисел, остатком от деления может быть не только целое, но и дробное число с числом знаков после десяти-

точного делителя, равным максимальному числу этих знаков у двух операндов. К таким типам данных применимы агрегатные функции и унарные операции.

Характеристики дробных чисел приведены в табл. 4.5.

Таблица 4.5. Дробные числовые типы данных

Тип данных	Размер (байт)	Диапазон значений
NUMERIC (<i>n</i> , <i>m</i>) DECIMAL (<i>n</i> , <i>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

4.3.3.1. Типы данных NUMERIC и DECIMAL

Типы данных NUMERIC и DECIMAL одинаковые, 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 — число знаков дробной части 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 число десятичных цифр, включающее число знаков целой и дробной части, — и *масштаб* (параметр *m*) — число 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).

Следовательно, выражение

```
DECIMAL
```

соответствует выражению

```
DECIMAL(18, 0)
```

Если вы укажете

DECIMAL(20)

то это означает задание

DECIMAL(20, 0)

Число знаков целой части может быть нулевым. Например, можно задать такое число:

DECIMAL(10, 10)

Число байтов, отводимых для хранения данных этого типа, зависит только от значения точности, как показано в табл. 4.6.

Таблица 4.6. Число байтов для типов данных NUMERIC и DECIMAL

Точность	Число байтов	Точность	Число байтов
1 – 10	5	21 – 29	13
11 – 21	9	30 – 38	17

4.3.3.2. 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 предназначены для работы с денежными (валютными) величинами.

Одно из отличий этих типов данных от типов данных NUMERIC и DECIMAL состоит в том, что при присваивании значений соответствующим элементам данных 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), ¢ (знак цента валюты евро, евроцент), \$ (доллар), € (евро), £ (фунт стерлингов), ¢ (цент "настоящий"), ¥ (иена) и ряд других.

При арифметических операциях 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 довольно сложный. Впрочем, характеристики остатка от деления тоже вычисляются не слишком просто.

4.3.3.3. 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 привести математические формулы. 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):

$$s_3 = \max(s_1, s_2)$$

4.3.3.4. Умножение

Для операции умножения (*) точность результата определяется суммой точностей операндов плюс единица. Масштаб результата — сумма масштабов умножаемых чисел. Или в принятых обозначениях:

$$p_3 = p_1 + p_2 + 1$$

$$s_3 = s_1 + s_2$$

4.3.3.5. Деление

Точность и масштаб результата деления двух чисел (/) определяются следующими формулами:

$$p_3 = p_1 - s_1 + s_2 + \max(6, s_1 + p_2 + 1)$$

$$s_3 = \max(6, s_1 + p_2 + 1)$$

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).

4.3.3.6. Остаток от деления

Наконец, характеристики результата операции получения остатка от деления (%) определяются следующими формулами:

$$p_3 = \min(p_1 - s_1, p_2 - s_2) + \max(s_1, s_2)$$

$$s_3 = \max(s_1, s_2)$$

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 ошибку типа переполнения. В частности, такая ошибка может возникнуть 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:

Сообщение 8115, уровень 16, состояние 2, строка 1

Ошибка арифметического переполнения при преобразовании expression к типу данных numeric.

4.3.3.7. 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 противоестественных 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 приведены в табл. 4.7.

Таблица 4.7. Числовые типы данных с плавающей точкой

Тип данных	Размер (байт)	Диапазон значений
<code>FLOAT (n)</code>	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}
<code>REAL</code>	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 число 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 первоначального большого числа.

Поскольку числа с плавающей точкой не являются точными, рекомендуется не использовать в операторах обращения к базе данных операций "равно" или "не равно". 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.

4.3.5.1. Агрегатные функции

К числовым данным (обычно к одному столбцу из нескольких строк таблицы) могут применяться агрегатные функции:

- ◆ `SUM()` — суммирует все значения списка;
- ◆ `MIN()` — отыскивает минимальное значение в списке;
- ◆ `MAX()` — отыскивает максимальное значение в списке;
- ◆ `AVG()` — вычисляет 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 `COUNT()`. Эта функция не обрабатывает числовых данных, а осуществляет подсчет числа повторений экземпляров некоторого объекта базы данных, обычно строк таблицы, отвечающих заданным условиям.

4.3.5.2. Тригонометрические функции

В языке Transact-SQL существует полный "джентльменский набор" тригонометрических функций, как прямых, так и обратных:

- ◆ **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, хотя никакие параметры функции не передаются.

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 RADIANT
-----

```

```

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 $\pi/2$.

4.3.5.3. Логарифмические функции, возведение в степень, извлечение корня

Вот полный набор таких функций:

- ◆ 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.

Соответствующая проверка использования функций приведена в *примере 4.3*.

Пример 4.3. Примеры обращения к логарифмическим функциям, функциям возведения 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 логарифмы 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 V(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.

4.3.5.4. Другие математические функции

Вот другие математические функции, которые не включены ни в одну из перечисленных групп:

- ◆ ABS(<арифметическое выражение>) — возвращает абсолютное значение заданного параметра, то есть 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>]) — округляет число 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) 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. При одном и том же заданном начальном значении функция 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 интересные из всех перечисленных 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 происходит несколько обращений к функции `RAND()`. Если функции передается одно и то же значение параметра, то она 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 как положительными, так 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`.
- ◆ Символьные строки в Юникоде (`Unicode Character Strings`). Типы данных `NCHAR`, `NVARCHAR` и `NTEXT`.

Для строковых данных допустимы все операции сравнения: `=`, `!=`, `<>`, `<`, `<=`, `!<`, `>`, `>=`, `!>`. Сравнение выполняется в так называемом лексикографическом порядке, при котором учитывается положение буквы в алфавите. При этом прописная буква и соответствующая ей строчная 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 вызове функции `fn_helpcollations()`. Присутствие в поле описания текста `"case-insensitive"` означает отсутствие чувствительности к регистру, т. е. символы `"А"` и `"а"` считаются равными. Напротив, текст `"case-sensitive"` означает, что символы чувствительны к регистру. Например, при сортировке `Cyrillic_General_CS_AI_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* существуют 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 число байтов, которое отводится для хранения строки. Точнее, это максимальное число *символов*, которое может содержать строка. Для любого строкового типа данных этот параметр может принимать значение от 1 до 8000. Для *VARCHAR* размер увеличивается на два байта.

Если при задании столбца указать *VARCHAR(max)*, то в нем можно хранить объем данных до 2 Гб. Данные *VARCHAR(max)* хранятся отдельно от данных таблицы. Для такого типа данных допустимы и обычные строковые операции.

Переменным строковым типам 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 число символов, то ничего страшного не произойдет. Не будет сгенерировано никакой ошибки; просто значение строки уменьшится до нужной величины. В некоторых СУБД 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.

ЗАМЕЧАНИЕ

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 (+).

Например, результатом конкатенации двух строк

```
'Smart ' + 'students'
```

будет строка

```
'Smart students'
```

Чуть позже мы рассмотрим и множество функций 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, кроме 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 катаканы и хираганы (принято в Японии).

Синтаксис задания типов данных:

```
NCHAR[ (<целое> ) ]
```

```
NVARCHAR[ (<целое> | max) ]
```

Для типа данных *NCHAR* есть синонимы *NATIONAL CHAR* и *NATIONAL CHARACTER*.

Для типа данных *NVARCHAR* возможна замена на 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 число символов, которое может содержать строка. Для типа данных *NCHAR* параметр может принимать значение от 1 до 4000. Для *NVARCHAR* размер увеличивается на два байта.

Если при задании столбца указать вариант *NVARCHAR(max)*, то, как и для типа данных *VARCHAR(max)*, в нем можно хранить большой объем данных.

В случае присваивания значения полю с данными Юникод строковые константы заключают в апострофы. Чтобы преобразовать строку 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 БД с типами данных **VARCHAR(MAX)**, **NVARCHAR(MAX)** и **VARBINARY(MAX)** могут хранить довольно большой объем данных — до $2^{31} - 1$ или 2 147 483 647 байтов, т. е. около 2 Гб. Если это книга в формате PDF, то в ней могут быть десятки тысяч страниц. Если это фильм довольно хорошего качества, то он может длиться часа полтора-два. При этом следует отметить, что для типа данных **VARBINARY(MAX)** в таблицах существует возможность назначить атрибут **FILESTREAM**, что позволяет снять ограничение на объем хранимых данных. Файловые потоки мы рассмотрим в следующей главе.

К данным этих типов могут применяться 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, задействовать разные строковые функции, то в процессе исследования существующих строковых функций SQL Server и дальнейшего их использования вы получите истинное удовольствие.

4.4.4.1. Определение размера **DATALENGTH()**, **LEN()**

Для определения длины (числа *байтов*) строки или строкового выражения можно использовать функцию **DATALENGTH()**, которой передается в качестве параметра исходное выражение. Другая функция, **LEN()**, возвращает число *символов* строки, за исключением конечных пробелов.

Еще раз подчеркну отличие этих функций. Функция **LEN()** возвращает число *символов*, а функция **DATALENGTH()** — число *байтов*. Иногда они отличаются друг от друга по значению.

Чтобы понять особенности применения строковых функций **LEN()** и **DATALENGTH()** по отношению к различным строковым типам данных в Management Studio, выполните операторы *примера 4.5*.

Пример 4.5. Применение 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 числе 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 клавиши `<Ctrl>+<T>`.

Результат выполнения операторов:

LEN	DATALENGTH
4	20
LEN	DATALENGTH
4	5
LEN	DATALENGTH
4	10
LEN	DATALENGTH
4	40

По результатам можно видеть, что функция `LEN()` действительно независимо от конкретного типа данных строковой переменной возвращает число символов, а не байтов, игнорирует конечные пробелы, но учитывает начальные пробелы.

Функция `DATALength()` возвращает число байтов в строке. Для строковых типов данных *переменной* длины учитывается только фактическое число символов, включая конечные пробелы. Для данных *фиксированной* длины всегда возвращается число символов, заданных при определении переменной.

4.4.4.2. Выделение 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 число первых (левых) символов строки. Синтаксис функции:

`LEFT(<исходная строка>, <число символов>)`

Функция `RIGHT()` возвращает указанное число правых символов строки. Синтаксис функции:

`RIGHT(<исходная строка>, <число символов>)`

Функция `SUBSTRING()` позволяет выделить любую часть строки — подстроку. Ее синтаксис:

`SUBSTRING(<исходная строка>, <начальная позиция>, <число символов>)`

Здесь из исходной строки выделяется указанное число 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 дасть сА 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 дасть сА вамъ	просите	сА вамъ	и дасть

4.4.4.3. Удаление 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(). Эти вложенные функции можно, разумеется, вызывать в любом порядке (пример 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.

4.4.4.4. Преобразование символов ASCII(), STR(), CHAR(), NCHAR(), UNICODE()

Эта группа функций позволяет получить код конкретного символа строки (число) или, наоборот, получить символ из заданного числа.

- ◆ ASCII(<строковое выражение>) — возвращает код (целое число) заданного первого символа в строке.
- ◆ STR() — преобразует число с плавающей точкой в строку символов. Синтаксис вызова функции:
STR(<числовое выражение> [, <длина> [, <число десятичных знаков>]])
Здесь <числовое выражение> — преобразуемое число (числовое выражение), которое может быть целым, дробным или с плавающей точкой. Параметр <длина> задает число символов, отводимых под результат. Учитывает цифры, десятичную точку, знак числа. Если длина не указана, то предполагается 10. Последний параметр, <число десятичных знаков>, задает число символов, отводимых для размещения дробных знаков преобразованного числа. Не может превышать 16.
- ◆ CHAR(<числовое выражение>) — преобразует число (целое) в символ. Параметром является целочисленное выражение.
- ◆ NCHAR(<числовое выражение>) — преобразует число (целое) в символ Юникода.
- ◆ UNICODE(<строковое выражение>) — возвращает код (целое число), соответствующего стандарту Юникода, первого символа в строке.

Использование функций ASCII(), CHAR(), NCHAR(), UNICODE() и STR() показано в *примере 4.9*.

Пример 4.9. Использование функций преобразования символов

```
USE master;
GO
SET NOCOUNT ON;
-- Объявление переменных
DECLARE @nstring nchar(13);
DECLARE @string char(13);
DECLARE @position int;
SET @nstring = N'я азъ глаголю';
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	a
109	m	1079	э
32		1098	ъ
115	s	32	
112	p	1075	г
101	e	1083	л
97	a	1072	a
107	k	1075	г
105	i	1086	о
110	n	1083	л
103	g	1133	ж

3.140 3.14 3.14159265

Вот этот пример давайте рассмотрим достаточно подробно, поскольку в нем содержатся некоторые новые для кого-то из вас 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'и азъ глаголю';
```

В скриптах SQL Server допустимы 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 нужно 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 единицу.

Так как обе строки имеют одинаковое число 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 функцию DATALENGTH() мы должны заменить функцией LEN():

```
WHILE @position <= LEN(@nstring)
```

Результаты выполнения функций ASCII(), CHAR(), NCHAR() и UNICODE() можно видеть в списке выведенных строк.

В последних строках пакета демонстрируется действие 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
```

4.4.4.5. Изменение регистра символов UPPER(), LOWER()

Функция UPPER(<строковое выражение>) переводит все буквы строки в верхний регистр, т. е. преобразует их в прописные или, как еще говорят, в заглавные, "боль-

шие". Функция применима к буквам как латинского, так и кириллического алфавитов. 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 никак не влияют.

Если вам интересно и у вас есть достаточно времени, то для исследований можете в предыдущий пример добавить вызов этих функций. Мне было интересно, и я выполнил для себя такие проверки.

4.4.4.6. Преобразование 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 дает возможность 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 символов, являющихся признаком завершения такой строки-идентификатора. Если параметр "ограничитель" отсутствует, то по умолчанию в качестве ограничителя для имени задаются квадратные скобки.

Например, выражение

```
QUOTENAME('asd[]fghj')
```

вернет значение

```
[asd[]fghj]
```

Здесь по умолчанию разделителем служат квадратные скобки. 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"
```

4.4.4.7. Поиск данных и изменение строки

`REPLICATE()`, `REVERSE()`, `REPLACE()`, `CHARINDEX()`

Функция `REPLICATE(<строковое выражение>, <целое>)` повторяет исходную строку указанное число 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>) заменяет 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, 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 заданным порядком сортировки — если порядок сортировки нечувствителен 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) = 'о';
DECLARE @string2 varchar(25) = ' Одинокий прохожий спешит';
SELECT CHARINDEX(@string1, @string2) AS 'Default collation',
       CHARINDEX('о' 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,	А роза упала на лапу Азора	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 `REPLACE()` в качестве входного параметра функции `REPLACE()`) и поиск несуществующей подстроки (`CHARINDEX`). В четвертом элементе выбора оператора `SELECT` функция `CHARINDEX()` возвращает значение 0, так как искомый текст отсутствует.

Во второй части примера проверяется наличие чувствительности к регистру у функции `CHARINDEX()` при выборе 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 различным порядком сортировки. В первом случае отсутствует чувствительность к регистру, и мы получаем результат 2, порядковый номер прописной буквы "О". Во втором случае задан порядок сортировки `Cyrillic_General_CS_AI_KS_WS`, являющийся чувствительным к регистру. Результатом будет число 6. Это номер первой в строке строчной буквы "о".

4.4.4.8. Поиск 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 существующие формы шаблонов. Дело в том, что ее можно использовать в операторе сложной выборки данных из одной или более таблиц БД, в операторе `SELECT`, в предложении `WHERE` в конструкции `LIKE`.

Синтаксис 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 возможностей функции (и ее сложность) заключается в шаблоне, который помимо обычных символов, используемых для поиска в строке, содержит специальные так называемые шаблонные символы, которые задают дополнительные условия выборки данных.

Шаблонные символы в функции `PATINDEX()` и в конструкции `LIKE` предложения `WHERE` оператора `SELECT` перечислены в табл. 4.8.

Таблица 4.8. Шаблонные символы

Символ	Значение
<code>%</code>	Произвольная строка, содержащая ноль или любое число символов
<code>_</code>	Любой одиночный символ
<code>[]</code>	Задание диапазона допустимых значений. Подходит любой символ из указанного диапазона. Символы диапазона указываются через дефис, например <code>[a-z]</code> . Это означает, что допустимыми являются все буквы латинского алфавита

Таблица 4.8 (окончание)

Символ	Значение
[^]	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('%O_O%', @S),
       PATINDEX('и НАЧ%', @S),
       PATINDEX('НАЧ%', @S),
       PATINDEX('%в_я%', @S),
       PATINDEX('%в[A-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 '%O_O%'. В шаблоне подряд идут два знака подчеркивания (не очень 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 нечувствителен 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 убедиться в наличии 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 соответствуют две подстроки: в словах "сворачивая" и "действия". Мы получаем число 40, т. е. первого слева появления нужной подстроки.

В следующем обращении к функции указан 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 так же из трех 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 строке, а к множеству строчковых данных из различных записей таблицы.

На этом рассмотрение строчковых функций закончим. На самом деле мы проигнорировали лишь две функции, которые связаны со звуковым преобразованием строк — SOUNDEX() и DIFFERENCE().

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 приведены 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

Таблица 4.9 (окончание)

Тип данных	Размер (байт)	Диапазон значений
TIME	3, 4 или 5	От 00:00:00.0000000 до 23:59:59.9999999
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 23:59:59.9999999
DATETIMEOFFSET	8, 9, 10	Дата от 1 января 0001 года до 31 декабря 9999 года. Время от 00:00:00.0000000 до 23:59:59.9999999. 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 — задает число дробных знаков секунд, или, как сказано в документации, число сотен наносекунд. Если масштаб не указан, то предполагается 7.

Тип данных DATETIME является объединением типов данных DATE и TIME с масштабом, который приблизительно равен 3.

Тип данных SMALLDATETIME 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>)]

Параметр <масштаб> — число от 0 до 7. Задает число дробных знаков секунд. Если не указан, то устанавливается 7.

Тип данных DATETIMEOFFSET похож на DATETIME2, при этом он еще позволяет учитывать и часовой пояс. Синтаксис задания типа данных DATETIMEOFFSET:

DATETIMEOFFSET[(<точность>, <масштаб>)]

Параметр <точность> — целое число от 26 до 34. Задает общее число знаков. Масштаб — число от 0 до 7. Параметр <масштаб> задает число 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

4.5.2.1. Задание даты и времени

Существует много способов задания даты.

В литерале, задающем дату, разделителями между номером дня, номером месяца и годом могут служить точка (.), знак минуса (-) или наклонная черта (/). Эти разделители взаимозаменяемые. В одном литерале возможен любой набор из разделителей, хотя этого одобрить 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 предусмотрена команда `SET DATEFORMAT`. Параметром команды могут быть `mdy`, `dmy`, `ymd`, `ydm`, `myd` и `dym`. 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 наиболее привычна 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()`. Ее синтаксис для преобразования даты в нужном формате:

```
CONVERT (VARCHAR, <преобразуемое выражение>, <стиль>)
```

Третий параметр функции `<стиль>` задает формат отображения. Вот некоторые варианты указания стиля:

- ◆ 20 или 120 — формат `гггг-мм-дд`;
- ◆ 4 — формат `дд.мм.гг`;
- ◆ 104 — формат `дд.мм.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 при помощи функции `DATEFROMPARTS`, которая появилась в версии SQL Server 2012. Ее синтаксис:

```
DATEFROMPARTS (<год>, <месяц>, <день>)
```

Варианты задания одной и той же даты, 14 января 2023 года, приведены в *примере 4.12*. Тут же даны и варианты вызова 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 = '14.01.2023';
SET DATEFORMAT mdy;
SET @D2 = '01-14-2023';
SET DATEFORMAT ymd;
SET @D3 = '2014/01/23';
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(2014, 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
2023-01-14	14.01.23	14.01.2023	2023-01-14
Стиль 1	Стиль 101	По умолчанию	
01/14/23	01/14/2023	2023-01-14	

Время всегда задается одинаковым образом в виде: `чч:мм:сс.пnnnnnn`. Здесь `чч` — часы, `мм` — минуты, `сс` — секунды, `пnnnnnn` — дробная часть секунд.

Если в одном литерале задается и дата, и время, то время отделяется от даты пробелом, причем можно ввести произвольное число пробелов. Например, чтобы указать дату 29 марта 2014 года и время 12 часов 16 минут 53 секунды, нужно написать:

```
'29.03.2014 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 число знаков после десятичной точки в дробной части времени. Он не может быть меньше, чем число 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 = '29.03.2014 12:16:53.0000021';  
SELECT CAST(@D1 AS VARCHAR(28)) AS 'Дата и время';  
SELECT DATETIMEFROMPARTS(2014, 03, 29, 12, 16, 53, 0000021)  
       AS 'DATETIMEFROMPARTS';  
SELECT DATETIME2FROMPARTS(2014, 03, 29, 12, 16, 53, 000021, 3)  
       AS 'DATETIME2FROMPARTS';  
GO
```

Результат:

```
Дата и время  
-----  
2014-03-29 12:16:53.0000021  
  
DATETIMEFROMPARTS  
-----  
2014-03-29 12:16:53.0000021  
  
DATETIME2FROMPARTS  
-----  
2014-03-29 12:16:53.021
```

Здесь выбран тип данных DATETIME2, который позволяет хранить и дату, и время. Обратите внимание, что для полноценного отображения результата в скрипте дата и время явно преобразуются в тип данных VARCHAR(28). Если не выполнить преобразование, то при отображении времени число 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 = '29.03.2023 12:16:53.1234567 +04:00';
SELECT @D0 AS 'Дата, время, смещение';
GO
```

Результат:

```
Дата, время, смещение
-----
2023-03-29 12:16:53.1234567 +04:00
```

4.5.2.2. Функции, возвращающие значения 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. Строго 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 число знаков после десятичной точки в дробной части времени. Не может быть меньше, чем число 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
-----
2023-02-25 21:06:41.8768980 2023-02-25 21:06:41.8768980
SYSDATETIMEOFFSET    VARCHAR
-----
2023-02-25 21:06:41.8778994 +04:00 2023-02-25 21:06:41.8778994 +04:00
SYSUTCDATETIME       VARCHAR
-----
2023-02-25 17:06:41.8778994 2023-02-25 17:06:41.8778994
CURRENT_TIMESTAMP    VARCHAR
-----
2023-02-25 21:06:41.877 фев 25 2023 9:06PM
GETDATE              VARCHAR
-----
2023-02-25 21:06:41.877 фев 25 2023 9:06PM
GETUTCDATE           VARCHAR
-----
2023-02-25 17:06:41.877 фев 25 2023 5:06PM
TIMEFROMPARTS
-----
12:16:53.0021
```

Здесь видны отличия в возвращаемых функциями результатах. Если в вашей работе часто приходится использовать дату и время, не поленитесь, подробно рассмотрите эти примеры.

4.5.2.3. 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 назначен английский язык. Чтобы задать русский язык, на котором будут выводиться тексты, нужно выполнить оператор

```
SET LANGUAGE N'Russian';
```

Можно записать оператор и в таком виде:

```
SET LANGUAGE N'русский';
```

В первом случае для указания языка используется псевдоним, алиас языка — англоязычное его название. Во втором варианте название языка записано 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 языков, поддерживаемых в SQL Server, с их характеристиками (краткие и полные названия месяцев, названия дней недели и ряд других) приведен 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>)
```

Параметр `<выделяемая часть>` определяет, какая часть данных будет возвращена функцией и в каком виде. Значения параметра показаны в *табл. 4.10*.

Таблица 4.10. Значение параметра `<выделяемая часть>` функции `DATENAME()`

Параметр	Сокращения	Возвращаемое значение
year	yy, yyyy	Четырехзначное значение года
quarter	qq, q	Номер квартала
month	mm, m	Название месяца
dayofyear	dy, y	Номер дня в году
day	dd, d	День в месяце
week	wk, ww	Номер недели в году
weekday	dw	Название дня недели

Таблица 4.10 (окончание)

Параметр	Сокращения	Возвращаемое значение
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

При вызове функции вы можете указать как полное название выделяемой части, так и сокращение.

Функция DATENAME() позволяет получить любой фрагмент даты и времени. Похожий результат дает функция DATEPART(), которая имеет такой же синтаксис, как и DATENAME():

DATEPART(<выделяемая часть>, <дата>)

Значения параметра <выделяемая часть> приведены 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 с помощью функций DATENAME() и DATEPART()

```
USE master;
GO
SET NOCOUNT ON;
SET DATEFORMAT dmy;
DECLARE @D DATE;
SET @D = '23.03.2014';
-- 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
-----
25                      2014
Quarter                 Day of Year  Week
-----
1                      82           13
Month Name              Weekday
-----
March                   Sunday
```

Параметры языка изменены на "русский".

LANGID LANGUAGE

```
-----
21     русский
Месяц                      День недели
-----
Март                      воскресенье
```

言語設定が 日本語 に変更されました。

LANGID LANGUAGE

```
-----
3      日本語
День недели
-----
日曜日
Смещение часового пояса
-----
```

+04:00

Changed language setting to us_english.

```
Day          Year
-----
23           2014
Quarter      Day of Year  Week
-----
1            82           13
Month Number Weekday
-----
3            1
```

Мы видим, что в зависимости от используемого языка названия месяца и дня недели выводятся в виде соответствующего текста. После изменения текущего языка системы мы отображаем и его характеристики — идентификатор и название.

Похожие результаты получены и при вызове функции `DATEPART()`.

В *примере 4.17* показаны обращения к функциям `DATENAME()` и `DATEPART()` для получения элементов времени.

Пример 4.17. Получение элементов времени при помощи 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
12	8	13

Millisecond	Microsecond	Nanosecond
725	725382	725382600

Hour	Minute	Second
12	8	13

Millisecond	Microsecond	Nanosecond
725	725382	725382600

Видно, что результаты выделения элементов времени, полученные при помощи этих двух функций, совершенно одинаковы.

4.5.2.4. 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>) — 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. Параметр <часть даты> задается фиксированным текстом в диапазоне от года до наносекунд. Подробности см. в Books Online.
- ◆ `EOMONTH` (<начальная дата> [, <добавляемые месяцы>]) — 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. Существуют 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})]
```

Параметр <размер> задает число байтов, которое отводится для хранения данных. Он может изменяться от 1 до 8000. Если задано ключевое слово `max`, то данные могут занимать до $2^{31} - 1$ байтов или около 2 Гб. Такой вариант использования двоичных данных мы уже упоминали чуть ранее, в подразд. 4.4.3. Если для хранения таких данных будут применяться файловые потоки, которые мы рассмотрим в следующей главе, то и этот размер может быть увеличен.

Этот тип данных позволяет хранить любые данные: тексты в различных форматах, изображения, звуки, видео, выполняемые файлы.

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.

В следующем разделе мы рассмотрим простой пример с `VARBINARY`, когда будем говорить о пространственных типах данных.

4.7. Пространственные типы данных

Пространственные (spatial) типы данных появились 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 местоположение в некоторой системе координат. В БД 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 и имеет прямое отношение к Евклидовой геометрии, где все объекты располагаются на плоской поверхности.

Тип данных `GEOGRAPHY` предназначен 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 полусферы (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.

Замечательная особенность поддержки пространственных типов данных в SQL Server — 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 представляет собой программную среду, которая устанавливается на компьютере вместе с операционной системой или с SQL Server.

Для внешнего представления пространственных типов данных SQL Server существует так называемый текстовый формат WKT (well-known text), определенный консорциумом OGC. В БД 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) вводятся понятия 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.

Задать значения столбцам таблиц и локальным переменным пространственных типов данных можно с помощью многочисленных методов 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 999999. 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 нулевое значение SRID.

Рассмотрим объекты этого типа данных.

4.7.1.1. Точка

Точка, объект 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 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 `Point`:

- ◆ `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 `Point` можно разбить на следующие группы:

- ◆ отображение характеристик объекта;
- ◆ создание объекта;
- ◆ дополнительные методы.

Отображение характеристик

Вот некоторые методы, позволяющие получить характеристики объекта `Point`:

- ◆ `STGeometryType()` — возвращает текст, отражающий тип геометрического объекта. Его можно применять для любого пространственного объекта. Для точки будет возвращена строка `"Point"`.
- ◆ `ToString()` — позволяет отобразить текст, соответствующий геометрическому объекту, в текстовом формате `WKT`.
- ◆ `STAsText()` — дает такой же результат, что и `ToString()`.
- ◆ `STAsBinary()` — позволяет отобразить текст, соответствующий геометрическому объекту, в двоичном формате `WKB`.

Создание объекта

Создать точку можно статическими геометрическими методами из пространства имен `.NET geometry`:

- ◆ `STPointFromText()`,
- ◆ `STGeomFromText()`,

- ◆ Point(),
- ◆ Parse(),
- ◆ STPointFromWKB(),
- ◆ STGeomFromWKB().

ЗАМЕЧАНИЕ

Пожалуй, выражение "создание объекта" в этом контексте не очень 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 встречается в русскоязычной литературе. 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 приведены в *листинге 4.2*. Обращению 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 параметр <текст WKT> должен содержать функцию задания геометрической точки. Он 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(0x0101000000000000000000000084000000000000001040, 0);
SET @P6 = geometry::STGeomFromWKB(0x0101000000000000000000000084000000000000001040, 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 создаются 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` (методы `ToString()` и `STAsText()`) и двоичное представление объекта в формате `WKB`.

ЗАМЕЧАНИЕ

При отображении строк в операторе `SELECT` в этом примере применена 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
-----
0x01010000000000000000000840000000000001040
```

ЗАМЕЧАНИЕ

Как вы наверное заметили, последняя строка в результатах этого примера содержит двоичные данные, значения которых были присвоены локальным переменным @P5 и @P6.

Дополнительные методы

Рассмотрим еще несколько методов, применимых к геометрическим точкам:

- ◆ `STEquals()` — сравнивает два геометрических объекта и возвращает значение 1, если оба объекта идентичны, т. е. совпадают их форма и местоположение. Иначе метод возвращает 0. Можно сравнивать только объекты с одинаковым значением идентификатора пространственной ссылки SRID.
- ◆ `STDistance()` — вычисляет расстояние между двумя точками на плоскости. Расстояние можно получить только между объектами с одинаковым значением идентификатора SRID. Этот метод также можно применить и к географическим объектам. Чуть позже мы с его помощью вычислим расстояние между различными городами на нашей планете.

- ◆ STLength() — определяет длину объекта. Для точки длина имеет значение 0. Более осмысленные результаты можно получить для других геометрических объектов.
- ◆ 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

Объект 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 он может быть замкнутым, когда начальная точка совпадает с конечной.

Для создания линии используются 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 предусмотрены те же методы, что и для точки:

- ◆ `STGeometryType()` — возвращает текст, отражающий тип геометрического объекта. В данном случае это строка `"LineString"`.
- ◆ `ToString()` и `STAsText()` — отображают текст, соответствующий геометрическому объекту в формате WKT.
- ◆ `STAsBinary()` — отображает текст в двоичном формате WKB.

Другие методы для `LineString`:

- ◆ `STEquals()` — сравнивает два геометрических объекта. Возвращает 1, если объекты равны, и 0, если не равны.
- ◆ `STDistance()` — вычисляет минимальное расстояние между двумя объектами на плоскости, то есть между двумя точками объектов, минимально удаленных друг от друга.
- ◆ `STDimension()` — возвращает размерность объекта. Для линии это число 1.
- ◆ `STLength()` — определяет общую длину объекта — сумму длин всех линий.
- ◆ `STStartPoint()` и `STEndPoint()` — возвращают текст в формате WKT, соответственно, начальной и конечной точки линии.

Получить идентификатор пространственной ссылки здесь также можно через 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 =
0x01020000000020000000000000000000F03F000000000000F03F000000000000F03F00000000000002240;
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 параметров ломаной линии. Объявлена также локальная переменная `@B` типа данных `VARBINARY`. Она служит 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 результа-

те полученный геометрический объект полностью совпадает с объектом, описанным в переменной @L1.

Созданные геометрические объекты приведены на *рис. 4.2*.

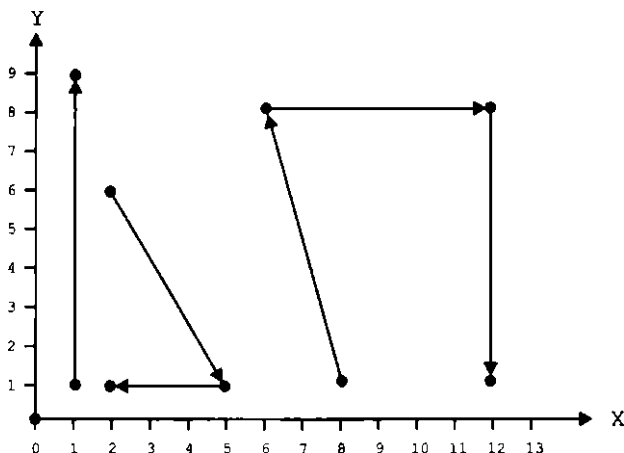


Рис. 4.2. Геометрические объекты типа "ломаная линия"

Стрелки на рис. 4.2 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

STAsBinary

```

-----
0x01020000000020000000000000000000F03F000000000000F03F000000000000F03F00000000000022
40
    
```

ToString	STAsText	STSrid
LINESTRING (1 1, 1 9)	LINESTRING (1 1, 1 9)	0

Result Of Comparison

```

-----
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 (объект `Polygon`) или многогранник является замкнутым объектом. Он должен содержать не менее трех точек. Чтобы объект был замкнутым, 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.

Создать полигон можно методами `STPolygonFromText()`, `Parse()`, `STGeomFromWKB()` и `STGeomFromText()`. При задании текста в формате WKT указывают ключевое слово "POLYGON".

Поскольку полигоны создаются замкнутыми ломаными линиями, для них можно вычислить площадь при помощи метода `STArea()`.

Для полигона существует 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 =
0x010300000001000000040000000000000000000000F03F00000000000000F03F00000000000000F03F0000000000
000840000000000000000400000000000000F03F00000000000000F03F00000000000000F03F
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 прямоугольный треугольник. Он совпадает с полигоном @P3. Второй — полигон @P2, прямоугольник. Стрелки на этом рисунке также отражают только порядок, в котором описывается создание объекта.

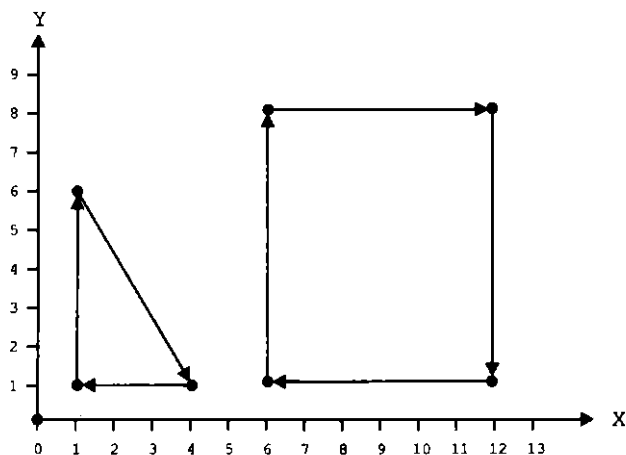


Рис. 4.3. Геометрические объекты "полигон"

Результат выполнения пакета:

STAsBinary

```
0x0103000000001000000040000000000000000000000F03F00000000000000F03F00000000000000F03F00000000000000840000000
000000004000000000000000F03F00000000000000F03F00000000000000F03F
```

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 (1.333333333333333357 1.666666666666666714)
```

После создания полигонов их вид отображается в двоичном и текстовом представлении. Рассчитывается 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.

- ◆ **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().
- ◆ **MultiPoint** — хранит коллекцию точек. Объект создают с помощью методов STGeomFromText(), STGeomFromWKB(), Parse() и STMPFromText().
- ◆ **MultiLineString** — хранит коллекцию ломаных линий. Для создания объекта служат методы STGeomFromText(), STGeomFromWKB(), STMLFromText(), STMLFromWKB(), Parse() и STMLStringFromText().

Отображение всех объектов можно выполнить с помощью 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`.

Результат выполнения пакета:

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. Этот тип данных, как и **GEOMETRY**, позволяет хранить данные о тех же семи типах объектов: `Point`, `MultiPoint`, `LineString`, `MultiLineString`, `Polygon`, `MultiPolygon` и `GeometryCollection`. Методы тоже похожи на методы (совпадающие по именам), которые применяются для объектов типа данных **GEOMETRY**. Только эти методы принадлежат пространству имен `geography`.

Вот некоторые методы, предназначенные для работы 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). Иначе говоря, 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 возвращает значение 0.
- ◆ `STDistance()` — наименьшее расстояние между двумя объектами.
- ◆ `STEndpoint()` — конечная точка экземпляра.

- ◆ `STEquals()` — сравнивает экземпляры. 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 и второму объекту.
- ◆ `STIntersects()` — возвращает 1, если экземпляры пересекаются, т. е. имеют общие точки, хотя бы одну общую точку. Иначе 0.

Характеристики объектов этого типа данных — 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. Вместо указания в координате минут и секунд значение градуса задается в виде дробного числа. Перевод из DMS в десятичное значение выполняется по формуле:

Десятичное представление координаты = градусы + (минуты / 60) + (секунды / 3600).

Поскольку координаты задаются с использованием типа данных `FLOAT`, максимальное число знаков в десятичном представлении не может превышать 15.

Например, для Саратова координаты: 51° 32' 26" северной широты, 46° 00' 31" восточной долготы. При переводе в десятичное представление широта будет 51.5405555555556, долгота 46.0086111111111.

Координаты Москвы: 55° 45' 08" северной широты, 37° 36' 56" восточной долготы. Десятичное значение: широта 55.7522222222222, долгота 37.6155555555556.

Земля представляет собой сплюснутый сфероид, причем очень неправильной формы. Для приближенного описания характеристик этого трехмерного геометрического объекта используют размер большой полуоси (расстояние от центра Земли до экватора) и размер малой полуоси (расстояние от центра Земли до полюса). В программировании географические объекты описывают с учетом размера большой полуоси и так называемого *улучшенного инвертированного отношения*, которое вычисляется по формуле:

Большая полуось / (большая полуось – малая полуось)

Идентификатор пространственной ссылки SRID определяет конкретную систему пространственных ссылок, характеристики которых необходимы для описания 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 около 400 различных пространственных ссылок. Они от-

личаются 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 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, приведены 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 — 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 в формате WKT представлено столбцом `well_known_text`. Текст не слишком наглядный. Разобьем его на несколько строк, добавив необходимые отступы:

```
GEOGCS
  ["Pulkovo 1995",
    DATUM
      ["Pulkovo 1995",
        ELLIPSOID
          ["Krassowsky 1940",
            6378245,
```

```

        298.3
    ]
],
PRIMEM["Greenwich", 0],
UNIT["Degree", 0.0174532925199433]
]

```

В самом начале указано, что идентификатор относится к географической системе GEOGCS. Затем приведено название: "Pulkovo 1995". После ключевого слова DATUM перечислены параметры. Текст "Pulkovo 1995" задает название. После ключевого слова ELLIPSOID указывается имя используемого эллипсоида: "Krassowsky 1940".

Затем даны числовые характеристики эллипсоида. Число 6 378 245 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 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 принят 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 величин и коэффициентов.

В *примере 4.24* приведен 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(55.752222222222, 37.615555555556, 4200);
DECLARE @Saratov AS GEOGRAPHY =
    geography::Point(51.540555555556, 46.008611111111, 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. Третий параметр — 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).

Результат выполнения пакета:

Широта Москвы	Долгота Москвы
-----	-----
55,752222222222	37,615555555556
Широта Саратова	Долгота Саратова
-----	-----
51,540555555556	46,008611111111
Расстояние Москва - Саратов	

725633,115488806	

Все расстояния указываются в тех единицах измерения, которые заданы в SRID, а это в нашем случае метры. Итак, 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 координаты городов представлены в системе 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 следующие: `SQL_VARIANT`, `TIMESTAMP`, `UNIQUEIDENTIFIER`, `HIERARCHYID`, `CURSOR`, `TABLE`, `XML`.

В документации говорится, что тип данных `TIMESTAMP` будет удален в следующих версиях системы, однако в версии 2022 он присутствует. 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 определен стандартами SQL.

4.8.1. Тип данных `SQL_VARIANT`

Тип данных `SQL_VARIANT` может хранить различные данные: целочисленные, строковые, пространственные и тип данных `XML`. Похожие типы данных существуют во многих скриптовых языках. Вот виды 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, вызвав функцию SQL_VARIANT_PROPERTY(). Функции передаются два параметра: имя переменной или столбца таблицы и параметр, задающий вид отображаемых метаданных. Параметр заключается в апострофы:

- ◆ BaseType — тип данных, хранящихся в настоящий момент в переменной.
- ◆ Precision — число знаков.
- ◆ Scale — число знаков после десятичной точки для числовых типов данных. Для всех остальных содержит 0.
- ◆ TotalBytes — число байтов для хранения данных и метаданных.
- ◆ Collation — порядок сортировки строковых данных. Для остальных типов данных возвращается значение NULL.
- ◆ MaxLength — максимальное число 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` предназначен 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 — весьма естественный способ 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 другие.

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.

Методы для работы с этим типом данных:

- ◆ `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 параметров.

ных 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 методу `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. Их можно заменить хорошо известной функцией `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 'H23 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

```

-----
0      /      0x

```

H21 Level H21 ToString H21 VARBINARY H21 Ancestor Bin H21 Ancestor Char

```

-----
1      /1/      0x58      0x      /

```

H22 Level H22 ToString H22 VARBINARY H22 Ancestor Bin H22 Ancestor Char

```

-----
1      /2/      0x68      0x      /

```

H23 Level H23 ToString H22 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 поддюжины локальных переменных типа данных `HIERARCHYID`. Мы создаем простую двухуровневую иерархию.

Переменной `@H1` присваивается значение корневого узла посредством метода `GetRoot()`. При отображении в первом операторе `SELECT` значения этой переменной видим, что она имеет уровень 0, текстовое (каноническое) ее представление `/`, значение в двоичном виде — `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);
```

Значение второго параметра, передаваемого методу, — `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.

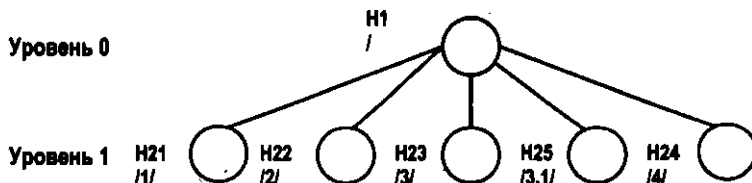


Рис. 4.4. Созданная иерархическая структура

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.

Вот результат:

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. Важный недостаток (с моей точки зрения) типа данных HIERARCHYID — фиксированное число 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. Следовательно, значения, полученные при помощи этой функции, будут уникальными на данном компьютере. Недостатком этой функции считается то, что получаемое при ее помощи значение можно предугадать, а это 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` мы продолжим 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) БД 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, применяемые для выборки данных из таблиц БД. 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 БД.

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 БД. 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. Объявление такой переменной осуществляется обычным оператором `DECLARE`, определяющим локальную переменную типа данных `CURSOR`.

Для объявления курсора как объекта предусмотрен оператор `DECLARE CURSOR`. Несколько сокращенный вариант синтаксиса объявления курсора приведен в *листинге 4.3*. 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 `@`.

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 БД `ALTER DATABASE`.

Варианты `FORWARD_ONLY` и `SCROLL` указывают, допустим ли при использовании курсора просмотр записей набора данных только вперед (`FORWARD_ONLY`) или возможен возврат к предыдущим записям (`SCROLL`), т. е. в обратном порядке. При задании `SCROLL` возможны все варианты навигации по набору данных в операторе `FETCH` (см. синтаксис этого оператора в листинге 4.4).

Вариант `READ_ONLY` определяет, что набор данных предназначен 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 изменять данные 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`, определяющий набор данных, с которым будет осуществляться деятельность через создаваемый курсор. Об операторе `SELECT` мы будем очень много говорить в главе 8. Сейчас же рассмотрим простой пример, в котором будет реализована уже известная нам форма этого оператора.

Во втором предложении `FOR` (в предложении `FOR UPDATE`) можно указать столбцы набора данных, которые можно изменять.

При работе с курсорами для навигации по набору данных, с которым связан курсор, применяется оператор `FETCH`. Его синтаксис приведен в листинге 4.4.

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>]... ];
```

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 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 число строк относительно текущей строки. Если задано положительное число, указатель 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 работы с набором данных, полученным 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 помощи курсора. Этот пример дублирует аналогичные действия и результат примера 3.2 из предыдущей главы.

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 БД) из системного представления каталогов `sys.databases`. При описании курсора указано ключевое слово `SCROLL`, значит, 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 оператором `OPEN`. В процессе открытия курсора выполняется соответствующий оператор `SELECT`, и в память загружается набор данных. После открытия курсор указывает на запись (несуществующую), предшествующую первой записи в полученном наборе данных. Для получения ссылки на следующую запись набора данных вызывается оператор `FETCH`. В необязательном предложении `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, который осуществляется при помощи оператора `WHILE`. В операторе задается условие продолжения цикла. Здесь используется системная функция `@@FETCH_STATUS`. Она станет 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 завершится.

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
master	1	2003-04-08	Cyrillic_General_CI_AS

NAME	ID	DATE	COLLATION
tempdb	2	2014-05-02	Cyrillic_General_CI_AS

NAME	ID	DATE	COLLATION
model	3	2003-04-08	Cyrillic_General_CI_AS

NAME	ID	DATE	COLLATION
msdb	4	2014-02-20	Cyrillic_General_CI_AS

NAME	ID	DATE	COLLATION
SimpleDB	5	2014-04-29	NULL

NAME	ID	DATE	COLLATION
BestDatabase	6	2014-05-01	Cyrillic_General_CI_AS

NAME	ID	DATE	COLLATION
Multy	7	2014-04-29	NULL

NAME	ID	DATE	COLLATION
MultyGroup	8	2014-04-29	NULL

NAME	ID	DATE	COLLATION
DBParam	9	2014-04-29	NULL

NAME	ID	DATE	COLLATION
ContainedDatabase1	11	2014-05-01	Cyrillic_General_CI_AS

NAME	ID	DATE	COLLATION
ContainedDatabase2	12	2014-05-01	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 `CURSOR`. 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 этого пакета осталась той же самой.

Присваивание локальной переменной указателя на ранее объявленный курсор осуществляется оператором `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 XML

Тип данных `XML` предназначен 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 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 применение в базах данных SQL Server.

XML — это язык разметки текстов. HTML также является языком разметки, однако включает 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 4.31).

Пример 4.31. 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 указывает 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 вложений. Здесь должна присутствовать строгая иерархия вложенности. Например, следующая конструкция будет неверной:

```
<T1>Некий текст<T2> Текст</T1>...</T2>
```

Существуют пустые теги, не содержащие никаких данных. Пустой тег можно указать двумя способами:

```
<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 тега, значения и конечного тега. Само "значение" также может быть элементом, т. е. включать в себя начальные теги, значения и конечные теги.

Чтобы добавить в документ примечания, их заключают между начальными `<!--` и конечными `-->` символами. Внутри размещается произвольный текст, который не проверяется анализатором.

Элементы в документе XML могут иметь атрибуты. В начальном теге элемента в этом случае указываются имя атрибута, знак равенства и — в кавычках — значение атрибута. Между элементами такой конструкции может добавляться любое число пробелов. Возможны варианты, когда элемент не имеет значения, все необходимые значения могут быть заданы атрибутами.

Аналогичный по смыслу документ, что и в *примере 4.31*, можно получить при использовании атрибутов (*пример 4.32*).

Пример 4.32. 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.

Результат выполнения пакета:

```
<Countries><Country ID="RUS" Name="Российская Федерация" /><Country ID="USA"
Name="United States of America" /></Countries>
```

В одном документе могут одновременно присутствовать как теги с заданными значениями, так и атрибуты. В *примере 4.33* показан такой случай.

Пример 4.33. 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 предусмотрены так называемые ссылки на сущности. Это пять конструкций:

- ◆ `&` — задает знак амперсанда.
- ◆ `"` — кавычка.
- ◆ `'` — апостроф.
- ◆ `<` — знак `<`.
- ◆ `>` — знак `>`.

Каждая из перечисленных конструкций заканчивается символом точка 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 `<`. Но здесь не должно быть подряд идущих трех символов: `]]>`.

Элемент с типом данных XML может быть типизированным или нетипизированным, он может содержать полный документ XML или его фрагмент.

Синтаксис задания типа данных XML для элемента (столбца таблицы или локальной переменной) показан в листинге 4.5.

Листинг 4.5. Синтаксис задания типа данных XML

```
XML [([CONTENT | DOCUMENT] <имя коллекции схем>)]
```

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, но 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.

4.8.5.1. Нетипизированные элементы XML

В *примере 4.34* показано создание простой таблицы, содержащей столбец с нетипизированным типом данных XML.

Пример 4.34. 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 4.35*.

Пример 4.35. 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>Краснодап</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

```

Данные, помещаемые в столбец XML, 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*.

	CODCTR	REGIONDESCR
1	RUS	<Regions><Region><ID>03</ID><Name>Краснодарский к...
2	USA	<Regions><Region><ID>AL</ID><Name>Alabama</Name>

Нажмите, чтобы открыть в редакторе XML.
Щелкните и удерживайте, чтобы выделить эту ячейку

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 выдаст более удобный для восприятия текст (*пример 4.3б*).

Пример 4.36. Пример вывода текста о регионах 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>
```

4.8.5.2. 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).

Создать коллекцию схем в текущей БД позволяет оператор `CREATE XML SCHEMA COLLECTION` (листинг 4.6).

Листинг 4.6. Синтаксис оператора `CREATE XML SCHEMA COLLECTION`

```
CREATE XML SCHEMA COLLECTION [<реляционная схема>.]
  <идентификатор SQL> AS <выражение>;
```

Здесь <выражение> является описанием, созданным средствами XSD.

Для добавления элементов в описание структуры документов XML в существующую коллекцию схем предусмотрен оператор `ALTER XML SCHEMA COLLECTION` (листинг 4.7).

Листинг 4.7. Синтаксис оператора ALTER XML SCHEMA COLLECTION

```
ALTER XML SCHEMA COLLECTION [<реляционная схема>.]
<идентификатор SQL> ADD <компонент схемы>;
```

Удалить существующую в БД коллекцию схем можно при помощи оператора DROP XML SCHEMA COLLECTION (листинг 4.8). 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 XML текущей БД используется системное представление каталогов sys.xml_schema_collections. Основные столбцы, возвращаемые этим представлением:

- ◆ 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 было показано в этом примере, нужно выполнить операторы, приведенные 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 БД. Для этого используется оператор SELECT, обращающийся к системному представлению каталогов sys.xml_schema_collections. Если коллекция схем уже существует в БД (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 содержать сведения о любом количестве ее регионов, где присутствуют 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	2014-02-20 20:48:35.617
65536	TestSchema	2014-05-02 22:19:22.880	2014-05-02 22:19:22.880
65537	TestSchemaCtr	2014-05-02 22:20:13.420	2014-05-02 22:20:13.420

(строка обработано: 3)

Список коллекций схем БД также можно получить, используя диалоговые средства SQL Server Management Studio. В окне **Обозреватель объектов** раскройте базу данных **BestDatabase**, затем папку **Программирование**, папку **Типы** и папку **Коллекция схем XML**. Полученный список показан на *рис. 4.6*.

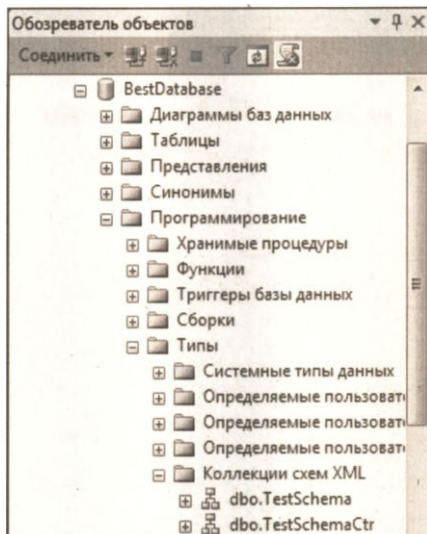


Рис. 4.6. Список коллекций схем

Чтобы просмотреть текст коллекции схем, нужно щелкнуть по ее имени правой кнопкой мыши и в появившемся списке контекстных меню выбрать элементы **Создать скрипт для коллекции схем XML | Используя CREATE | Новое окно редактора запросов**. Появится новое окно, которое будет содержать текст.

Создайте таблицу REFCTXXML, 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 REFCTXXML
( CODCTR      CHAR(3) NOT NULL,      /* Код страны */
  REGIONDESCR XML(TestSchemaCtr), /* Описание регионов */
  CONSTRAINT PK_REFCTXXML 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 REFCTXXML (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

```

4.8.5.3. 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 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 помощью 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
```

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`. Применять 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 предусмотрен оператор `CREATE TYPE`. Его синтаксис для создания псевдонимов и табличных типов данных приведен в листинге 4.9.

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>)

```

Оператор 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 БД везде, где допустимы системные типы данных.

Табличные типы данных могут служить 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, число 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. Назначить в качестве базового типа ранее созданный пользовательский тип нельзя.

В качестве базового возможен 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 недопус-

тимостью пустого значения, для типов данных DECIMAL и NUMERIC дает возможность также указать масштаб и точность, а для строковых типов данных — число хранимых символов. И все.

4.9.3. Создание псевдонима в диалоговых средствах Management Studio

Для создания псевдонима в Management Studio в окне **Обозреватель объектов** щелкните правой кнопкой мыши по элементу **Определяемые пользователем типы данных** и в контекстном меню выберите элемент **Создать определяемый пользователем тип данных**. Появится диалоговое окно. Создадим псевдоним для типа данных BIT и назовем его BOOLEAN — *рис. 4.7*. Из выпадающего списка **Тип данных** выберем тип данных **bit**.

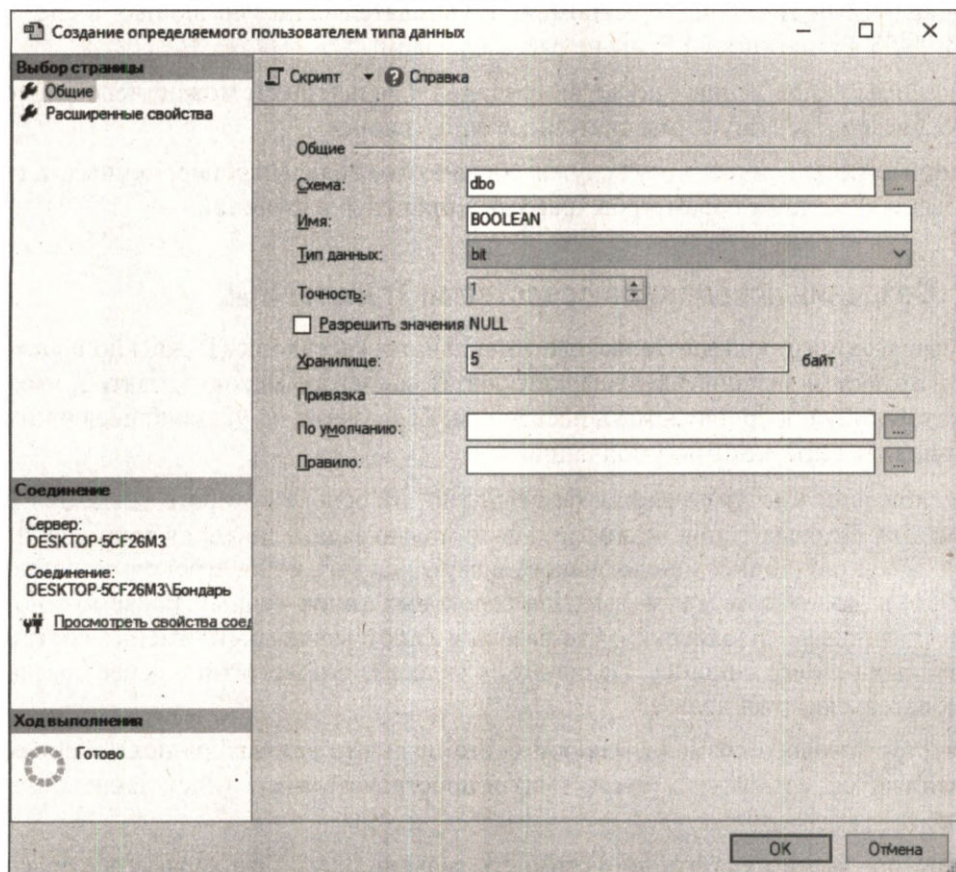


Рис. 4.7. Создание псевдонима BOOLEAN

После щелчка по кнопке **ОК** в базе данных BestDatabase будет создан новый псевдоним.

4.9.4. Создание пользовательского табличного типа данных средствами Transact-SQL

Если создается пользовательский табличный тип 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 число вычисляемых столбцов и ограничений таблицы. Элементы описания структуры таблицы отделяются друг от друга запятыми.

Здесь мы относительно кратко рассмотрим создание табличного типа данных и необходимые для этого средства. Подробно о таблицах поговорим в следующей главе 5.

4.9.4.1. Определение столбца

Имя столбца должно быть уникальным среди имен столбцов этого табличного типа данных. Для столбца должен быть указан тип данных. Это может быть системный или пользовательский тип данных, созданный ранее в этой БД. Если столбец является строковым (`CHAR` или `VARCHAR`), то для него в предложении `COLLATE` может быть указан порядок сортировки, отличный от установленного по умолчанию.

Ключевые слова `NULL` и `NOT NULL` задают допустимость для столбца неизвестного значения. Если не указано, то действуют 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 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.

4.9.4.2. Ограничения столбца

Для столбца можно задать так называемые *ограничения*. При описании табличного пользовательского типа данных допустимы три ограничения: 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 истины, чтобы в таблицу была записана новая строка (`INSERT`) или чтобы были выполнены изменения данных строки (`UPDATE`).

4.9.4.3. Вычисляемые столбцы

В таблице могут присутствовать вычисляемые 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 БД, а получается, "вычисляется", при выборке данных из таблицы. Однако при задании ключевого слова `PERSISTED` значение такого столбца будет храниться в таблице БД.

Для вычисляемых столбцов, как и для обычных, можно задавать 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 можно описать следующим образом:

```
NETSALARY AS SALARY * 0.87,
```

Что интересно, вычисляемый столбец может к тому же являться первичным или уникальным ключом.

4.9.4.4. 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. На-

пример, 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.

4.9.4.5. 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 псевдонима. Если такие псевдонимы уже существуют в вашей БД, то не следует создавать их заново. Затем создается табличный тип данных. При описании столбцов в качестве их типов данных служат только что созданные псевдонимы.

Тут же мы применили на практике вычисляемый столбец из предыдущего примера. В последней строке описания таблицы задается ограничение первичного ключа на уровне таблицы.

Обратите внимание, что после операторов создания псевдонимов записан оператор GO. Это необходимо сделать, чтобы вновь создаваемые псевдонимы стали "видимы-

ми" для последующих операторов в этом скрипте и их можно было использовать в создаваемом далее табличном типе данных.

Чтобы в Management Studio просмотреть список пользовательских типов данных в конкретной БД нужно в **Обозревателе объектов** раскрыть список баз данных (**Базы данных**), раскрыть нужную БД, раскрыть папку **Программирование**, раскрыть **Типы** и раскрыть **Определяемые пользователем типы данных**. Чтобы увидеть список пользовательских *табличных* типов данных, нужно в папке **Типы** раскрыть еще и папку **Определяемые пользователем табличные типы**. 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 **Обозревателя объектов** выглядит описание созданных типов данных и табличного типа данных для пустой БД.

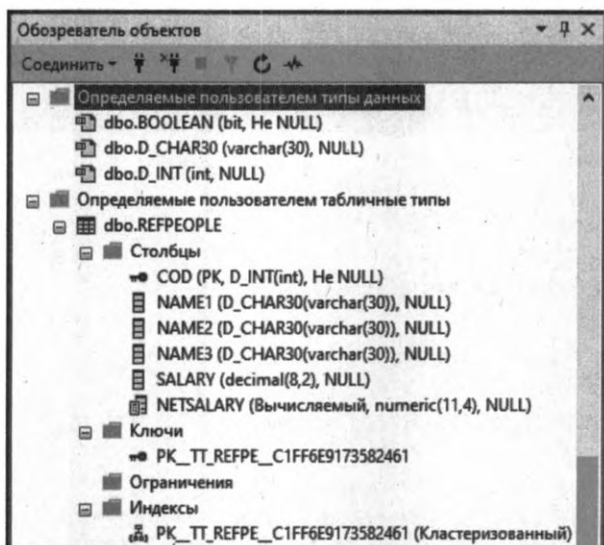


Рис. 4.8. Описание пользовательских типов данных и табличного типа данных

Для псевдонимов в этом списке помимо имени отображаются базовый тип данных и допустимость пустого значения.

В случае табличного пользовательского типа данных можно раскрыть папку со списком столбцов созданного нами типа данных (**Столбцы**), папку ключей (**Ключи**), папку ограничений (**Ограничения**) и папку индексов (**Индексы**). Кстати, на *рис. 4.8* мы видим, что для ограничения первичного ключа система по умолчанию создает кластерный индекс с длинным и нудным названием. В дальнейшем при создании реальных таблиц нашей БД мы любым ограничениям будем присваивать довольно осмысленные имена, чтобы при различных вариантах отображения сведений о таблицах получать читаемые тексты.

В скриптах создания демонстрационной БД существует множество примеров псевдонимов. После выполнения полного скрипта создания базы `BestDatabase` и пользо-

вательских типов данных в Management Studio можно увидеть, что все типы принадлежат схеме dbo, так как 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 пользовательский табличный тип данных, мы в окне **Обозревателя объектов** раскрываем список баз данных, переходим к БД BestDatabase, открываем папку **Программирование**, раскрываем **Типы**, щелкаем правой кнопкой мыши по строке **Определяемые пользователем табличные типы** и выбираем в контекстном меню элемент **Создать определяемый пользователем тип таблицы**. 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.

4.9.6. Удаление пользовательского типа данных

Для удаления пользовательского типа данных из БД в языке Transact-SQL есть оператор DROP TYPE. Его синтаксис показан в *листинге 4.10*.

Листинг 4.10. Синтаксис оператора DROP TYPE

```
DROP TYPE [<имя схемы>].<имя типа данных>;
```

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 **Обозреватель объектов** правой кнопкой мыши по имени псевдонима и в контекстном меню выберите элемент **Удалить**. Появится окно удаления объекта (рис. 4.9).

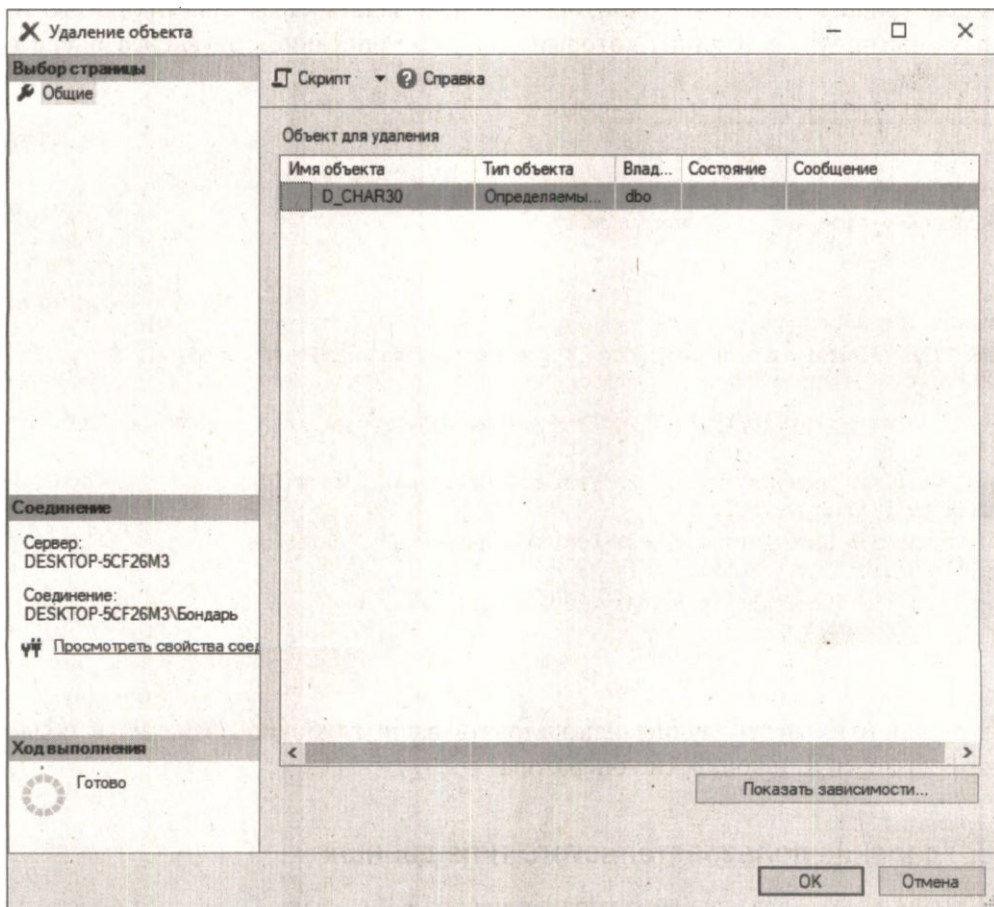


Рис. 4.9. Окно удаления объекта

Если здесь щелкнуть по кнопке **ОК**, то в следующих диалоговых окнах вы сможете увидеть, почему удаление невозможно. Чтобы получить подробную информацию о зависимостях этого псевдонима, нужно в окне **Удаление объекта** щелкнуть мы-

шью по кнопке **Показать зависимости**. Появится окно со списком объектов, зависящих от данного псевдонима (рис. 4.10).

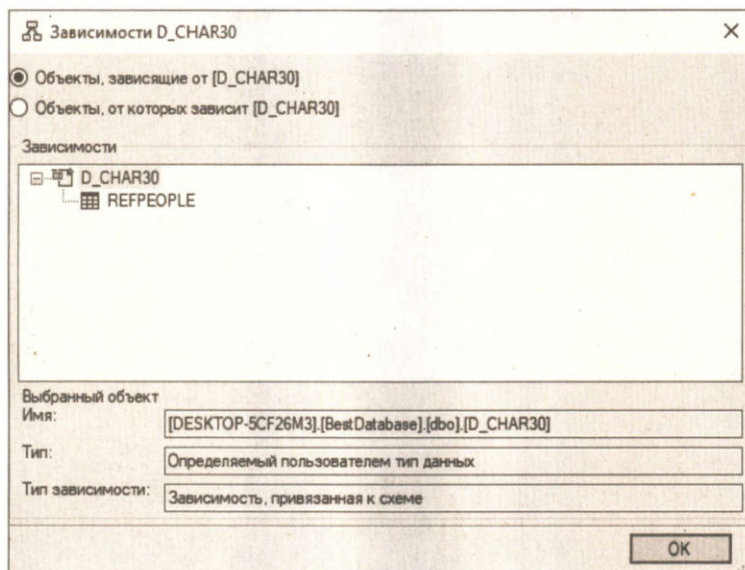


Рис. 4.10. Объекты, зависящие от псевдонима

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 щелкнуть по радиокнопке **Объекты, от которых зависит [D_CHAR30]**.

Чтобы удалить пользовательский тип данных (да и любой другой объект БД), 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 следующей главе.

Что дальше?

В следующей главе мы рассмотрим создание таблиц, основных объектов реляционной БД, 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 почти все возможности, используемые при создании таблиц.

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 диалоговыми средствами Management Studio.
- ◆ Добавление, удаление, изменение характеристик обычных и вычисляемых столбцов.
- ◆ Добавление, изменение и удаление ограничений.

Важнейший объект базы данных — это таблица. В таблицах содержатся все обрабатываемые данные исходной предметной области.

С точки зрения пользователя структура таблицы — это некоторое число 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 столбца 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 число строк. Она может быть 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 предусмотрен оператор `CREATE TABLE`. Его синтаксис приведен в листинге 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>] ...) ] ;
```

В версии SQL Server 2014 появились таблицы, оптимизированные для памяти, которые позволяют сильно повысить производительность системы обработки данных. Эти таблицы могут иметь только первичные ключи. Связи между такими таблицами отсутствуют. Синтаксис создания этих таблиц отличается от обычного. В данной книге мы такие средства рассматривать не будем.

5.1.1. Общие характеристики 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 БД — в той, которая была установлена в последнем операторе `USE`. Если не задано имя схемы, то таблица создается в схеме базы данных по умолчанию — обычно это схема `dbo`, однако при создании нового пользователя можно назначить 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. Последние 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 при 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 потребляется сетевой трафик.

Число 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.

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.

Предложение FILESTREAM_ON. Для хранения данных файловых потоков (filestream) это предложение задает имя файловой группы, указывает файловую группу по умолчанию или схему секционирования для распределения данных файловых потоков, если таблица является секционированной.

В необязательном предложении WITH можно указать параметры таблицы. Большинство параметров появилось еще в версии SQL Server 2012:

```
<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>
}
```

Вариант DATA_COMPRESSION задает режим сжатия данных и позволяет указать секции, для которых будет применяться сжатие. Допустимы режимы:

- ◆ NONE — сжатие данных всей таблицы или в указанных секциях отсутствует.
- ◆ 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.

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 этой главе 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 применен 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 сгенерирует 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 сгенерирует 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 сгенерирует имя.

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 Кб. При этом для хранения данных BLOB используются отдельные страницы, размер таких данных может быть очень большим. Для строковых данных переменной длины система также реализует 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 применяется стратегия хранения, сокращающая объем требуемой памяти. Таблицы, содержащие разреженные столбцы, называются "широкими таблицами" (wide table). Такие таблицы могут содержать до 30 000 столбцов, однако число неразреженных столбцов в них не должно превышать 1024.

Синтаксис определения столбца таблицы приведен в *листинге 5.2*.

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. Параметры столбца

Имя столбца должно быть уникальным в данной таблице. Другие таблицы базы данных могут содержать столбцы с теми же именами.

Тип данных — системный или пользовательский тип данных, созданный ранее для этой БД. Столбец таблицы не может иметь тип данных `TABLE`. Подробности о типах данных см. в предыдущей главе 4.

Необязательное ключевое слово `FILESTREAM` допустимо только для типа данных `VARBINARY (MAX)`. Все значения такого столбца будут храниться не в самой БД, 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.

Необязательное предложение `COLLATE` позволяет для строкового столбца задать порядок сортировки, отличный от порядка, установленного для всей базы данных.

Ключевые слова `NULL | NOT NULL` определяют допустимость для столбца значения `NULL`. По умолчанию при стандартных установках системы допустимо наличие значения `NULL`. Для столбцов, входящих в состав первичного ключа таблицы, рекомендуется всегда явно задавать характеристику `NOT NULL` независимо от установок системы.

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`.

5.1.2.2. Параметр `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`). Еще их называют столбцами идентификаторов. Подобные столбцы, как правило, служат для задания искусственных первичных ключей таблицы.

По умолчанию для первой добавляемой строки таблицы этому столбцу будет установлено значение 1. Все последующие присваиваемые значения будут увеличи-

ваться 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 число дробных знаков должно быть указано нулевым. В таблице допускается только один столбец типа `IDENTITY`.

В обычных условиях столбцу с этой характеристикой нельзя явно присвоить значение. Его имя не должно присутствовать в операторах `INSERT` и `UPDATE`. При этом существует оператор `SET IDENTITY_INSERT`, позволяющий задавать конкретное значение такому столбцу. Его синтаксис приведен в листинге 5.3.

Листинг 5.3. Синтаксис оператора `SET IDENTITY_INSERT`

```
SET IDENTITY_INSERT [[<имя базы данных>.]<имя схемы>.]<имя таблицы>
{ ON | OFF };
```

Если для таблицы указано `SET IDENTITY_INSERT ON`, то столбцу `IDENTITY` можно явно присвоить значение. Если вновь присваиваемое значение больше текущего, которое автоматически присваивается этому столбцу, то новое значение заменит текущее (автоматически присваиваемое) значение.

В одной сессии соединения с базой данных только для одной таблицы БД можно установить такую возможность.

5.1.2.3. 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, функции для работы с ним и примеры приведены в предыдущей главе 4. Такой столбец обязательно должен быть описан в таблице при использовании файловых потоков.

5.1.2.4. SPARSE

Ключевое слово `SPARSE` указывает, что столбец разреженный. 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 таблицы. Задать ограничения одного столбца можно также и на уровне таблицы.

5.1.3. Ограничения столбца и ограничения таблицы

В базах данных существуют четыре вида ограничений как для одного столбца, так и для группы столбцов таблицы:

- ◆ 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 приведен в *листинге 5.4*. "Начальный" 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>  
  | <уникальный ключ>  
  | <внешний ключ>  
  | <ограничение 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. Или когда получите сообщение об ошибке, имеющей отношение к конкретному ограничению. 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 моих разработках встречаются не очень часто. Для внешнего ключа применяю FK<номер ключа в таблице>_<имя таблицы> и для ограничения CHECK — конструкцию CN<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 ключей

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 попытке поместить такую же вторую запись появится 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 ключей 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 приведен в *листингах 5.5 и 5.6* соответственно.

Листинг 5.5. Синтаксис ограничения первичного ключа на уровне столбца таблицы

```
<первичный ключ (столбец)> ::=  
    PRIMARY KEY [ CLUSTERED | NONCLUSTERED ]  
    [ WITH (<параметр индекса> [, <параметр индекса>]...) ]
```

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>]...) ]
```

Синтаксические конструкции для отдельного столбца (на уровне столбца) и для группы столбцов, т. е. на уровне таблицы, мало отличаются друг от друга. Основным 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 не нужно. Любое огра-

ничество 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 а не 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 названием вида ключа. Соответствующие синтаксические конструкции приведены в листингах 5.7 и 5.8.

Листинг 5.7. Синтаксис ограничения уникального ключа на уровне столбца таблицы

```
<уникальный ключ (столбец)> ::=
    UNIQUE
    [ CLUSTERED | NONCLUSTERED ]
    [ WITH (<параметр индекса> [, <параметр индекса>]...) ]
```

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>]...) ]
```

В рассмотренных синтаксических конструкциях создания первичного и уникального ключей описывается и предложение `WITH`, где в скобках можно перечислить некоторые параметры индекса, создаваемого для поддержания ключа.

Создаваемые индексы

Для первичного и уникального ключей система автоматически создает индекс.

Вы можете указать, что для поддержки ключа будет использоваться кластерный (ключевое слово `CLUSTERED`) индекс. Употребляется также 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 с иерархической структурой. 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 — важнейший элемент механизма, обеспечивающего связи между таблицами в базе данных, реализующего отношение "один ко многим". Это отношение является универсальным отношением, поддерживающим все основные связи в реляционных БД.

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, приведен в *листингах 5.9 и 5.10*.

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 } ]
```

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 } ]
```

Рассмотрим основные отличия этих синтаксических конструкций:

- ◆ Для внешнего ключа на уровне столбца таблицы ключевые слова `FOREIGN KEY` могут отсутствовать. На уровне таблицы их указание 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 только на первичный ключ этой таблицы.

ЗАМЕЧАНИЕ

В очередной раз хочу напомнить, что в большинстве случаев не следует ориентироваться на значения по умолчанию, в том числе для предложений `ON DELETE`, `ON UPDATE`, необходимо задавать конкретные характеристики и синтаксические выражения. Скрипты пишем в первую очередь для себя, чтобы нам было точно понятно, что должно быть создано.

Важно, что структура внешнего ключа и типы данных столбцов, входящих в его состав, у дочерней таблицы должны полностью совпадать (вплоть до числа 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 отсутствие каких-либо действий (значение по умолчанию). Если удаляется строка родительской таблицы, для которой существует некоторое количество подчиненных строк дочерней таблицы (значение внешнего ключа в этих строках дочерней таблицы совпадает со значением первичного или уникального ключа удаляемой строки родительской таблицы), то произойдет нарушение декларативной целостности данных в БД. 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. Основным инструментом для восстановления целостности данных — триггеры.
- ◆ `CASCADE` — приводит к удалению всех соответствующих подчиненных строк дочерней таблицы, т. е. 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 далее в этой главе.

- ◆ **SET DEFAULT** — устанавливает в значение по умолчанию все столбцы внешнего ключа дочерней таблицы. Наблюдая на практике такое 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 при задании предложения **ON DELETE**:

- ◆ **NO ACTION** — означает отсутствие каких-либо 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 изменены данные 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. Пример задания синтаксиса в простом варианте иллюстрирует *листинг 5.11*.

Листинг 5.11. Синтаксис определения ограничения **CHECK**

```
<ограничение CHECK> ::=  
CHECK (<логическое выражение>)
```

Вообще говоря, никаких особых формальных требований к содержимому логического выражения не предъявляется. Вы, например, можете задать какое-либо усло-

ние, 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, создаются 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 БД, а вычисляться только при отображении данных таблицы.

Синтаксис определения вычисляемого столбца показан в листинге 5.12.

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>)

```

Основные свойства вычисляемого и обычного столбца весьма похожи. Главный момент — 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`), 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. При удалении можно указать лишь параметры `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 (`SPARSE`) 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. В таблице допускается только один набор столбцов.

Использование набора столбцов может повысить производительность системы.

Давайте сейчас пока приостановим дальнейшее рассмотрение других конструкций в синтаксисе оператора `CREATE TABLE` и разберем примеры создания достаточно простых таблиц.

5.2. Примеры простых 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 */
  DESCR   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 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 назначен тип данных `VARCHAR`. Два последних столбца, карта страны и дополнительное описание могут хранить графиче-

ку 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 реализуем 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 удаляем. Для этих целей используется системное представление каталогов `sys.tables`, при помощи которого проверяется существование (функция `EXISTS()`) таблицы с именем `REFREG`.

Однако если в БД есть объекты, зависящие от данной таблицы, то такую таблицу удалить нельзя, пока не будут устранены 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 дальше в этой главе, в *подразд. 5.7.1*.

Пример 5.2. Таблица регионов

```
USE BestDatabase;
GO

  /*** Справочник регионов ***/
IF EXISTS (SELECT * FROM sys.tables
           WHERE NAME = 'REFREG')
  DROP TABLE REFRREG;
GO
CREATE TABLE REFRREG
( 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 REFRCTR (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 регионы, относящиеся к этой стране. 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, пригодной 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 */
  DESCR 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, входящие во внешний ключ дочерней таблицы, полностью соответствуют характеристикам столбцов первичного ключа родительской таблицы.

Созданная иерархия показана на *рис. 5.1*.

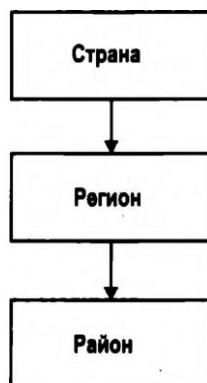


Рис. 5.1. Трехуровневая иерархия административно-территориального деления стран

ЗАМЕЧАНИЕ

Мне приходилось наблюдать, как разработчики БД добавляют в подобную структуру 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, оба равны 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 присутствуют два вычисляемых столбца — `FULLNAME` и `NET_SALARY`. Причем их значения не хранятся в БД, 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 чем 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 с помощью той же операции конкатенации.

Следует заметить, что эта таблица плохо подходит для большинства систем обработки данных реальной жизни. Здесь мы ее рассматриваем только в качестве простой иллюстрации создания таблиц. Чуть позже приведем примеры более правильных решений.

Часто в задачах обработки данных нужно описывать различные организации и некоторые их характеристики. В *примере 5.5* создана таблица организаций, которая может быть полезной 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 применяется 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 организации 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 коды страны, региона и района. Нужно сказать, что в некоторых случаях структуризация адреса получается не слишком 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

```

Полагаю, 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 понадобится 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*.

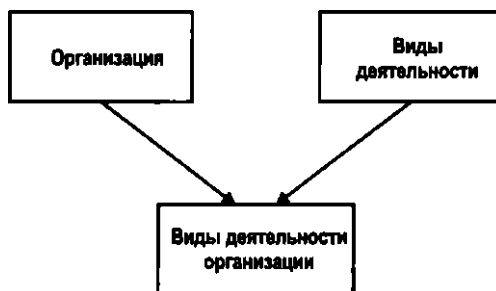


Рис. 5.2. Задание видов деятельности организаций

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 встречались оба варианта. Я так и не оценил преимущества какого-либо из них.

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 (и продолжается несколько лет) одна ну очень грустная история. Итак, по порядку.

Несколько упрощенный (и неверный в текущей версии SQL Server) вариант задания таблицы, хранящей сведения о людях, приведен в *примере 5.7*.

Пример 5.7. Таблица людей (неверная в SQL Server 2014)

```

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 */
  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 SET NULL,
  CONSTRAINT FK2_PEOPLE
    FOREIGN KEY (CODFATHER) 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 первичный ключ *той же самой* таблицы, но на различные ее строки.

Полужирным шрифтом выделены предложения, которые не нравятся системе. Это конструкции

```
ON DELETE SET NULL
```

Мы получаем сообщение, что подобные указания могут вызвать заикливание или многочисленные каскадные изменения в системе. Вообще-то наше описание весьма логично и не должно вызывать никакого заикливания 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 выдает то же самое сообщение.

Подобная структура у меня давно 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 релизов. Однако вышедшая версия SQL Server 2022 все равно выдает 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 2014)

```
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 */
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

```

Задача сохранения целостности данных решается, это можно выполнить при использовании триггеров, о которых нам предстоит разговор в *главе 11*.

Здесь мы также видим простой пример ограничения CHECK:

```
CONSTRAINT CH_PEOPLE CHECK (SEX IN ('0', '1')),
```

В логическом выражении ограничения присутствует 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**. Следовательно, 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 понадобится третья связующая таблица (*пример 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 */
  CODPEOPLE 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 на *рис. 5.3*.

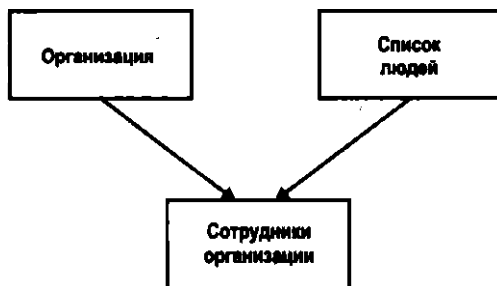


Рис. 5.3. Описание сотрудников организаций

В реальных системах, где нужны довольно подробные сведения о людях, часто указывают адреса (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, система все равно продолжит функционировать, 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). Напомним, синтаксис этого предложения выглядит следующим образом:

```
ON { <схема секционирования> (<разделяющий ключ>)
    | <файловая группа>
    | "default"
}
```

Это предложение определяет, где должны располагаться конкретные строки таблицы. Существуют 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 присутствует 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

ей 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 БД, они принадлежат всей базе данных.

Синтаксис оператора создания схемы секционирования `CREATE PARTITION SCHEME` приведен в листинге 5.13.

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 } ] ... );
```

Имя схемы секционирования должно быть уникальным в текущей базе данных.

ЗАМЕЧАНИЕ

Обратите внимание, что здесь употребляется термин *scheme*, 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 *scheme*. Во втором случае речь идет об относительно самостоятельной части БД. Вообще слово *schema* было введено в терминологию баз данных комитетом CODASYL где-то в 60-е годы прошлого столетия. Правда, означало оно совсем не то, что подразумевается под схемой БД в SQL Server, — описание структуры *всей* базы данных. Описание *части* БД для в конкретной задачи называлось подсхемой.

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 *секциям*, описанным в соответствующей функции секционирования, на которую указывает ссылка в операторе создания схемы секционирования.

Давайте сразу же рассмотрим средства удаления и изменения схемы секционирования. Удалить существующую схему секционирования позволяет оператор `DROP PARTITION SCHEME` (листинг 5.14).

Листинг 5.14. Синтаксис оператора удаления схемы секционирования

```
DROP PARTITION SCHEME <имя схемы секционирования>;
```

Чтобы вы могли удалить схему секционирования, на нее не должны ссылаться никакие объекты БД. Например, не должно быть таблиц, использующих ее имя в предложении `ON`. Перед удалением такой схемы необходимо устранить в базе данных все ссылки на нее.

Внести изменения в существующую схему секционирования можно с помощью оператора `ALTER PARTITION SCHEME` (листинг 5.15).

Листинг 5.15. Синтаксис оператора изменения схемы секционирования

```
ALTER PARTITION SCHEME <имя схемы секционирования>  
NEXT USED [ <файловая группа> ] ;
```

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 приведен в листинге 5.16.

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>]...]);
```

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 применена операция "меньше или равно", т. е. в диапазон попадут строки, у которых разделяющий ключ меньше или равен указанной границе и больше предыдущего значе-

ния 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. Число 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 предусмотрен оператор `DROP PARTITION FUNCTION` (листинг 5.17).

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` (листинг 5.18).

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 тем задачам, с которыми вы можете когда-нибудь столкнуться в 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, Hugeshard. Эта организация имеет множество 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, Азии, Африке и Австралии. Антарктида пока не присутствует в этом списке.

Каждому центральному офису был присвоен порядковый номер: Америке — 1, Европе — 2, Азии — 3, Африке — 4 и Австралии — 5.

Для повышения производительности системы обработки данных по персоналу были использованы только что рассмотренные средства секционирования таблицы. Все сведения по персоналу, относящемуся к одному центральному офису (а это множество региональных офисов с большим числом 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 сведений о сотрудниках 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 БД — 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 БД 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 осмысленные значения.

ЗАМЕЧАНИЕ

В приведенном примере все файлы размещаются на одном диске и в одном и том же каталоге. Разумеется, это сделано лишь для иллюстрации использования средств секционирования таблицы в соответствии 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 носителях, 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 БД Hugehard. Нам нужны 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 БД 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	1024	-1
2	1	LOG	HugehardLog	ONLINE	1024	268435456
3	0	ROWS	America	ONLINE	1024	-1
4	0	ROWS	Europe	ONLINE	1024	-1
5	0	ROWS	Asia	ONLINE	1024	-1
6	0	ROWS	Africa	ONLINE	1024	-1
7	0	ROWS	Australia	ONLINE	1024	-1

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 БД Hugehard.

Пример 5.12. Отображение файловых групп БД 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

Мы видим, что помимо первичной файловой группы в базе данных присутствуют 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 оптимальна. Однако в данном случае это подходящее решение, потому что эта фирма категорически запрещает ее сотрудникам работать еще в каких-либо других организациях, так что все характеристики сотрудника можно описывать в одной таблице.

Здесь в состав первичного ключа нам пришлось ввести как его часть, так и столбец REGION. Система секционирования требует, чтобы разделяющий ключ был частью первичного или уникального ключа. Для этого столбца мы задали 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, собственно 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 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 поподробнее рассмотрим всю картину секционирования строк таблицы. В предложении FOR VALUES указаны *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 последовательно создать функцию и схему секционирования (*примеры 5.15 и 5.14*), а также таблицу сотрудников организации (*пример 5.13*).

Пожалуй, более правильный вариант создания всех объектов БД показан в *примере 5.16*. Здесь все выполняется в нужном порядке и предусмотрены все подходящие проверки существования объектов БД.

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 проверка и удаление 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 применяется представление просмотра каталогов `sys.partition_schemes`, а для проверки существования функции секционирования — представление `sys.partition_functions`, при вызове которых указываются имена соответствующих объектов БД.

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 — параметр циклов заполнения таблицы. В одном цикле строки помещаются 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 **Обозреватель объектов** раскройте сначала БД Hugerhard, затем папку таблиц и далее созданную таблицу **dbo.STAFF**.

Если раскрыть список столбцов этой таблицы, то можно увидеть перечень всех ее столбцов с основными их характеристиками.

Раскрыв папку **Ключи**, можно увидеть список всех ключей таблицы. В нашем случае это только первичный ключ **PK_STAFF**.

Если раскрыть папку **Индексы**, 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 **Ограничения** мы видим имя ограничения таблицы. Это ограничение на значение столбца **REGION**.

Если раскрыть папку **Хранилище**, а затем **Схемы секционирования**, то в окне будут показаны схемы секционирования базы данных. Если раскрыть папку **Функции секционирования**, то можно увидеть список функций секционирования.

Некоторые из этих характеристик можно увидеть на *рис. 5.4*.

ЗАМЕЧАНИЕ

Похожие средства секционирования используются не только для строк таблиц, но 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 элементы **Создать скрипт для таблицы** | **Используя CREATE** | **Новое окно редактора запросов**.

В новом окне появится скрипт создания таблицы **STAFF**. Фрагмент этого скрипта приведен в *примере 5.18*.

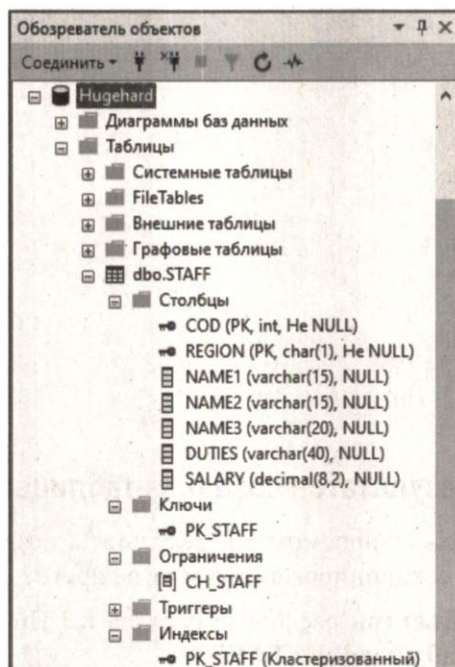


Рис. 5.4. Отображение характеристик таблицы в окне **Обозреватель объектов** Management Studio

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.

Однако, тот факт, что таблица является секционированной, тут увидеть нельзя.

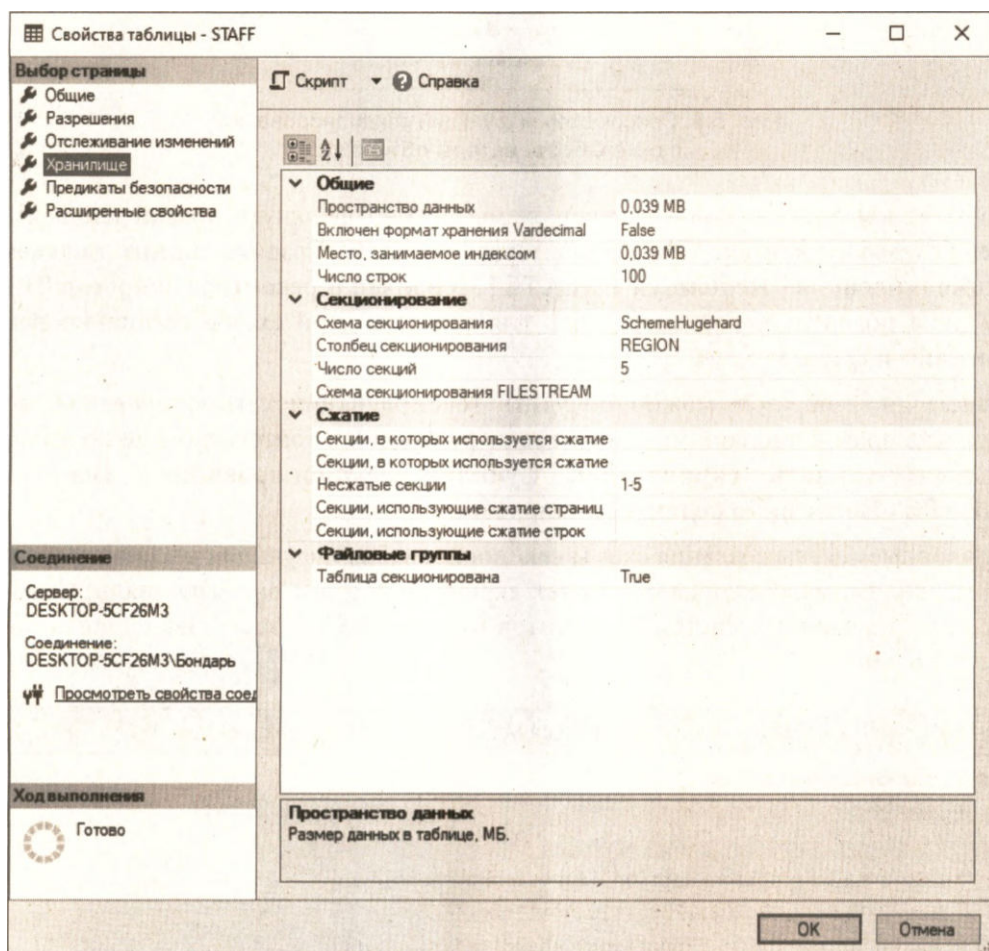


Рис. 5.5. Отображение характеристик секционированной таблицы в Management Studio

Чтобы выяснить наличие секционированности таблицы, щелкните правой кнопкой по имени таблицы и в контекстном меню выберите строку **Свойства**. Появится окно, отображающее некоторые свойства таблицы. В левой верхней части этого окна щелкните мышью по строке **Хранилище**. Теперь в основном окне будут отображаться характеристики, связанные с хранением таблицы. В частности, указано, что таблица секционированная (**Таблица секционирована** имеет значение **True**), отображено имя схемы секционирования, число разделов секционирования (**Число секций**) и разделяющий ключ (**Столбец секционирования**) — *рис. 5.5*.

Чтобы увидеть список схем секционирования и функций секционирования, в окне **Обозреватель объектов** раскройте папку **Хранилище**, далее раскройте папки **Схемы секционирования** и **Функции секционирования** (*рис. 5.6*)

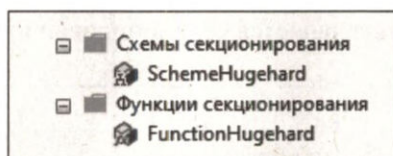


Рис. 5.6. Список схем и функций секционирования в окне **Обозреватель объектов**

Чтобы просмотреть исходный текст схемы, нужно щелкнуть правой кнопкой по имени схемы и в контекстном меню выбрать элементы **Создать скрипт для схемы секционирования | Используя CREATE | Новое окно редактора запросов**. 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 элементы **Создать скрипт для функции секционирования | Используя CREATE | Новое окно редактора запросов**.

Полученные тексты создания схемы секционирования и функции секционирования, как мы видим, мало отличаются от тех скриптов, которые мы выполняли. См. тексты сгенерированных системой скриптов в *примере 5.19* (здесь два скрипта объединены в один).

Пример 5.19. Сгенерированные скрипты создания схемы и функции секционирования таблиц

-- Функция секционирования

USE [Hugehard]

GO

/***** Object: PartitionFunction [FunctionHugehard]

Script Date: 27.02.2023 20:51:26 *****/

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: 27.02.2023 20:53:12 *****/
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** решила открыть представительства и в Антарктиде. Поэтому появилась потребность выполнить некоторые изменения в БД.

В первую очередь в базу данных нужно добавить новую файловую группу, в которую следует поместить один файл. В *примере 5.20* приведены 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 Antartctica;
GO
ALTER DATABASE Hugehard
ADD FILE
( NAME = Antartctica,
  FILENAME = 'D:\Hugehard\Antartctica.ndf')
TO FILEGROUP Antartctica;
GO
```

Поскольку в систему был добавлен еще один регион, следует изменить ограничение **CHECK** в таблице **STAFF**, добавив в список допустимых значений столбца **REGION** строковое значение **6**, которое будет соответствовать региону Антарктида. Для этого 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 первоначально использованного для таблицы STAFF. Там мы применяли конструкцию IN. Смысл ограничения остался точно таким же.

Теперь нужно изменить сначала схему, а затем 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. Вторым оператор добавляет в функцию секционирования новое значение разделяющего ключа.

Добавьте в таблицу персонала сведения об антарктических сотрудниках, 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 БД и саму базу данных в первую очередь средствами 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 задокументированы в хорошо написанных комментариях, позволяет достаточно легко вносить изменения как в сам процесс разработки, так и в результат документирования этой разработки.

Также существуют диалоговые средства создания объектов базы данных, которые можно использовать в процессе работы с Management Studio. Иногда при разработке так называемых макетных проектов бывает довольно удобно применять именно диалоговые средства 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 Hugelhard. В окне **Обозреватель объектов** раскройте базу данных **Hugelhard**. Щелкните правой кнопкой мыши по папке **Таблицы** и в контекстном меню выберите элемент **Создать | Таблица**. Появится окно, где можно описывать столбцы создаваемой таблицы и их характеристики (рис. 5.7).



Рис. 5.7. Описание столбцов создаваемой таблицы

Создадим таблицу, аналогичную таблице сотрудников STAFF, с полным повторением ее структуры и порядка секционирования и назовем ее STAFF1.

В поле **Имя столбца** введите имя COD. Из выпадающего списка **Тип данных** выберите тип данных int. Снимите отметку в поле **Разрешить значения NULL**, чтобы не допустить помещения в столбец значения NULL, поскольку столбец будет входить в состав первичного ключа создаваемой таблицы.

Теперь нужно установить для столбца свойство IDENTITY. В нижней части формы, в окне **Свойства столбца** (рис. 5.8), раскройте свойство **Спецификация идентификатора**, из выпадающего списка подсвойства (**Идентификатор**) выберите **Да**. Начальное значение идентификатора и шаг приращения идентификатора уже уста-

новлены в единицу, как и в случае описания таблицы STAFF, что нас вполне устраивает. Так что здесь ничего больше менять не будем.

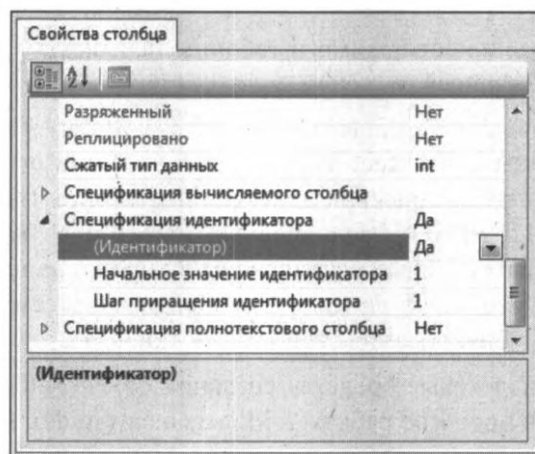


Рис. 5.8. Характеристики столбца COD создаваемой таблицы

Создайте описания остальных столбцов таблицы. Тип данных выбирается из выпадающего 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.

Имя столбца	Тип данных	Разрешить значения NULL
COD	int	<input type="checkbox"/>
REGION	char(1)	<input checked="" type="checkbox"/>
NAME1	varchar(15)	<input checked="" type="checkbox"/>
NAME2	varchar(15)	<input checked="" type="checkbox"/>
NAME3	varchar(15)	<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

Теперь нужно задать первичный ключ таблицы. Он должен состоять из двух столбцов: REGION и COD. Щелкните правой кнопкой мыши по столбцу REGION и в контекстном меню выберите элемент **Задать первичный ключ**. 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 **Индексы и ключи**. Появится окно, содержащее описание всех индексов таблицы (рис. 5.10).

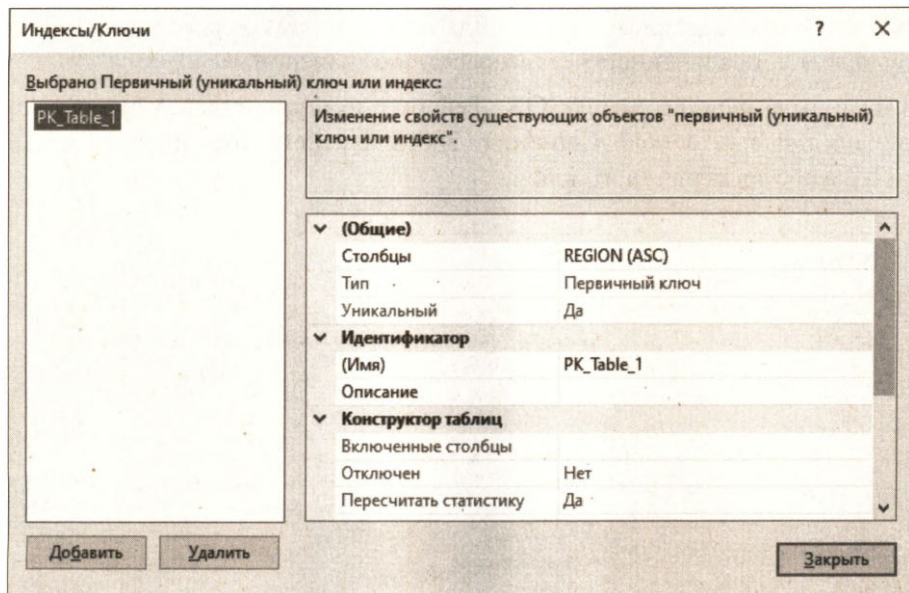



Рис. 5.10. Список индексов создаваемой таблицы

Здесь сразу можно изменить имя индекса первичного ключа, набрав в поле **(Имя)** (имя этого поля заключается в скобки) текст PK_STAFF1. Чтобы добавить к первичному ключу еще один столбец, нужно щелкнуть мышью по полю **Столбцы**. Затем в правой части поля — по появившейся кнопке с многоточием .

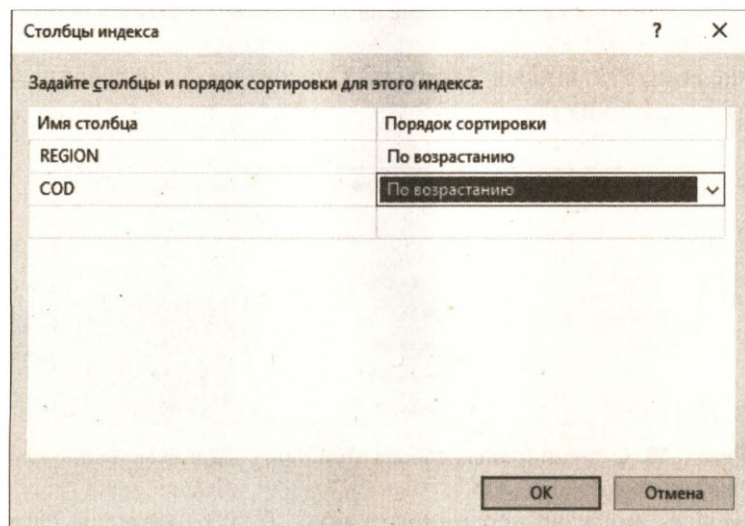


Рис. 5.11. Окно столбцов индекса

После этого в следующем появившемся окне **Столбцы индекса** (рис. 5.11) нужно добавить в список столбец COD, щелкнув мышью по пустой строке ниже поля REGION

и выбрав его из выпадающего списка. Для обоих столбцов, входящих в состав индекса, выбрана упорядоченность по возрастанию значений.

Щелкните в этом окне по кнопке **ОК**. Теперь структура индекса будет такой, как показано на *рис. 5.12* в поле **Столбцы**. Здесь изменено имя индекса и добавлен столбец **COD** в состав первичного ключа.

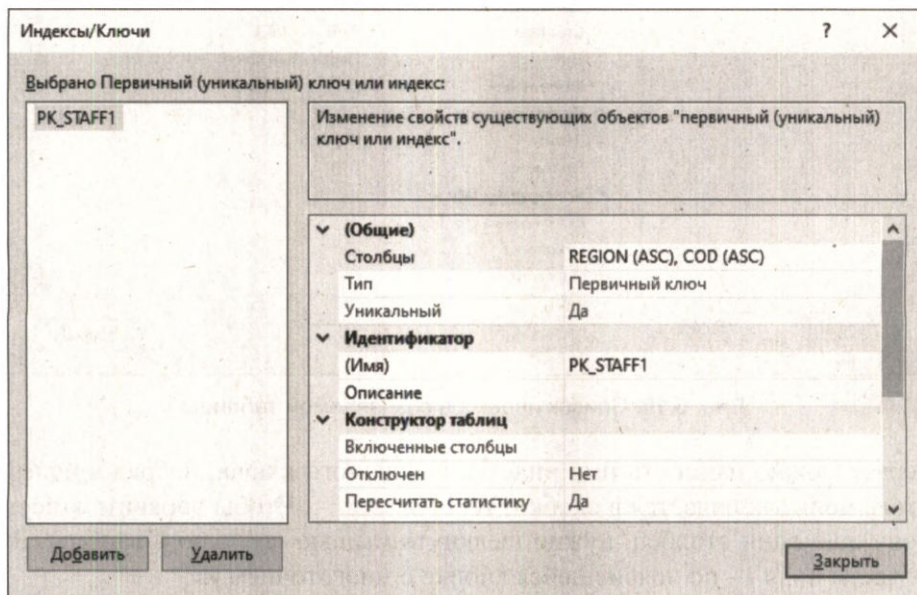


Рис. 5.12. Измененные характеристики индекса

В этом окне щелкните по кнопке **Заккрыть**. Список столбцов таблицы теперь будет выглядеть, как показано на *рис. 5.13*.

	Имя столбца	Тип данных	Разрешить значения NULL
🔑	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. 5.13. Окончательный список столбцов создаваемой таблицы

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 **Сохранить Table_1**.

Появится окно задания имени таблицы (рис. 5.14), где нужно ввести STAFF1 и щелкнуть мышью по кнопке **ОК**.

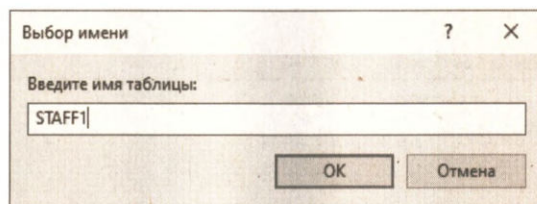


Рис. 5.14. Ввод имени таблицы

Таблица будет сохранена с этим именем в базе данных.

Чтобы задать необходимые характеристики секционирования таблицы, нужно в окне **Обозреватель объектов** раскрыть таблицы базы данных **Hugehard**, щелкнуть правой кнопкой мыши по имени только что созданной таблицы **STAFF1** и выбрать в меню **Хранилище | Создать секцию**. Появится начальное окно Мастера создания секционирования (рис. 5.15).

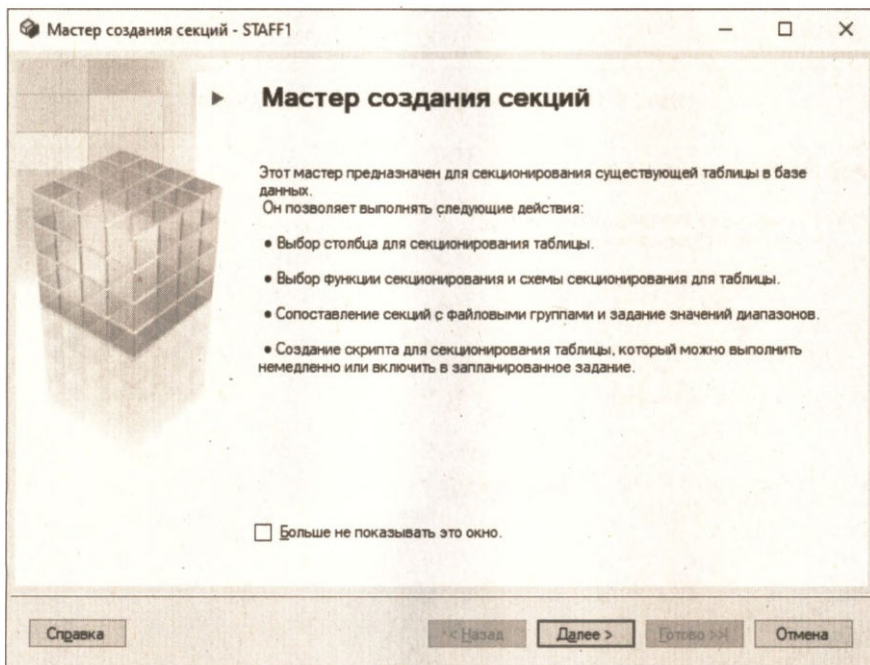


Рис. 5.15. Начальное окно Мастер создания секций

Щелкните по кнопке **Далее**.

В следующем окне (рис. 5.16) нужно выбрать столбец, который будет использован в качестве разделяющего ключа.

Отметьте столбец **REGION** и щелкните мышью по кнопке **Далее**.

Мастер создания секций - STAFF1

Выбор столбца секционирования

Выберите столбец, по которому будет секционироваться таблица.

Доступные для секционирования столбцы:

Имя столбца	Тип данных	Длина	Точность	Масштаб
<input type="radio"/> COD	int	4	10	0
<input type="radio"/> DUTIES	varchar	40	0	0
<input type="radio"/> NAME1	varchar	15	0	0
<input type="radio"/> NAME2	varchar	15	0	0
<input type="radio"/> NAME3	varchar	20	0	0
<input checked="" type="radio"/> REGION	char	1	0	0
<input type="radio"/> SALARY	decimal	5	8	2

☐ Выровнять эту таблицу с выделенной секционированной таблицей:

☐ Выровнять хранение всех не уникальных и уникальных индексов с индексированным столбцом секционирования

В верхней сетке содержатся столбцы секционирования для выбранной таблицы. Выберите столбец, который будет использоваться в качестве столбца секционирования в этой таблице.

Справка < Назад Далее > Готово >> Отмена

Рис. 5.16. Выбор столбца секционирования

Мастер создания секций - STAFF1

Выбор функции секционирования

Создайте новую функцию секционирования или выберите существующую.

Выбор функции секционирования

☐ Новая функция секционирования

☒ Существующая функция секционирования: FunctionHugehard

Справка < Назад Далее > Готово >> Отмена

Рис. 5.17. Выбор функции секционирования

В следующем окне (рис. 5.17) нужно отметить радиокнопку **Существующая функция секционирования**, поскольку мы собираемся использовать ранее созданную функцию секционирования. В правой части этого поля присутствует имя функции: FunctionHugehard.

Щелкните мышью по кнопке **Далее**.

В следующем окне (рис. 5.18) похожим образом мы выбираем существующую в базе данных схему секционирования, отметив соответствующую радиокнопку и выбрав из выпадающего списка справа нужное имя схемы. На самом деле имя этой единственной схемы уже указано в данном поле.

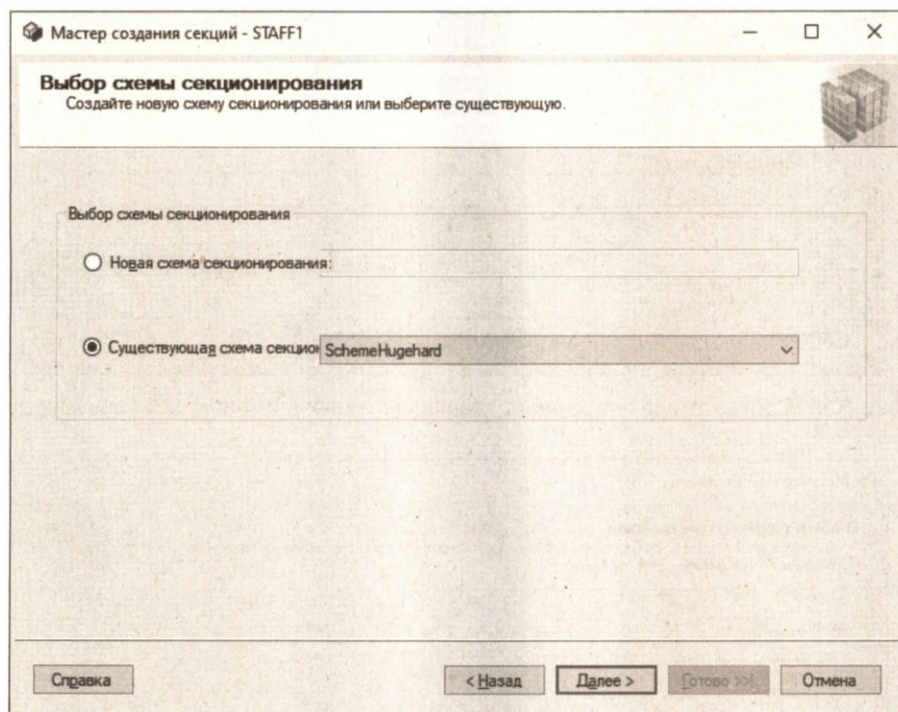


Рис. 5.18. Выбор схемы секционирования

После щелчка по кнопке **Далее** появляется окно, в котором описывается соответствие значений разделяющего ключа именам файловых групп (рис. 5.19).

Здесь ничего менять не нужно, просто щелкните мышью по кнопке **Далее**.

Следом появится окно (рис. 5.20), в котором можно выбрать действия с полученным скриптом. Его можно запустить немедленно, сохранить в файл (**Вывести скрипт в файл**) или вывести в новое окно запросов.

Выберем последний вариант и щелкнем мышью по кнопке **Далее**.

Щелчок по кнопке **Далее**. В следующем окне даются итоговые сведения об используемых объектах базы данных (рис. 5.21).

Здесь только нужно щелкнуть мышью по кнопке **Готово**.

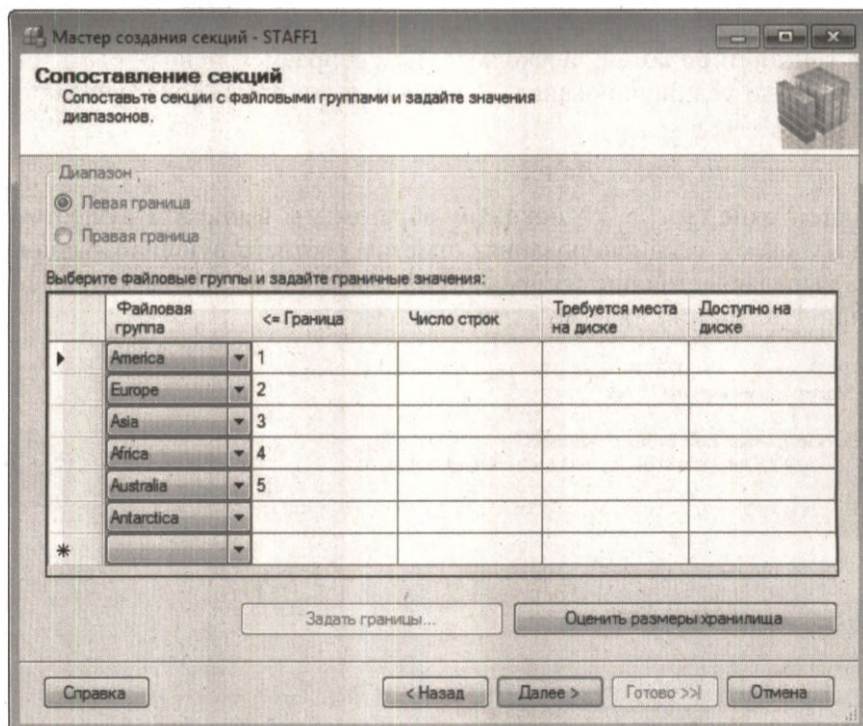


Рис. 5.19. Соответствие значений разделяющего ключа именам файловых групп

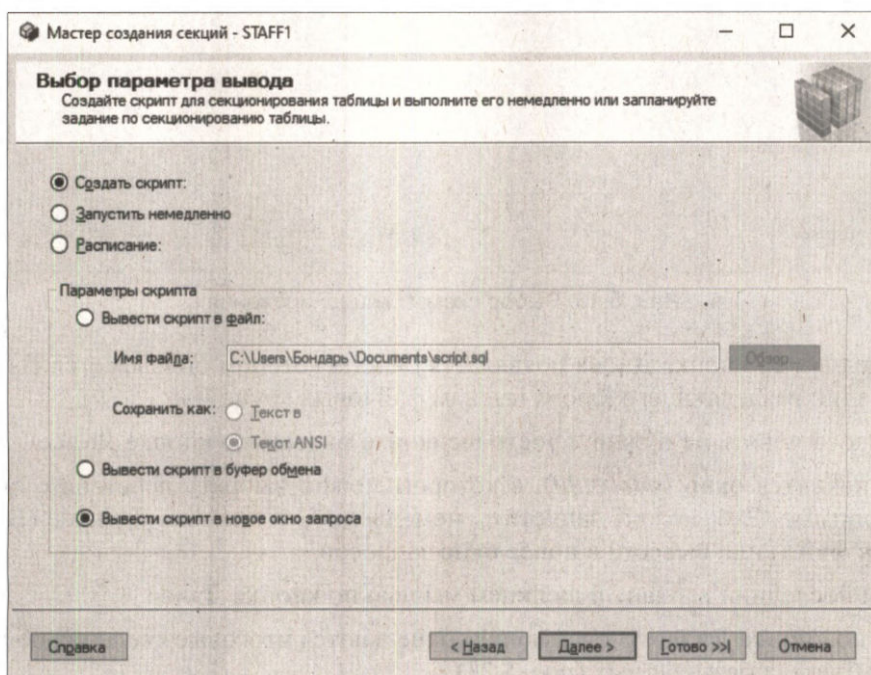


Рис. 5.20. Сохранение созданного скрипта

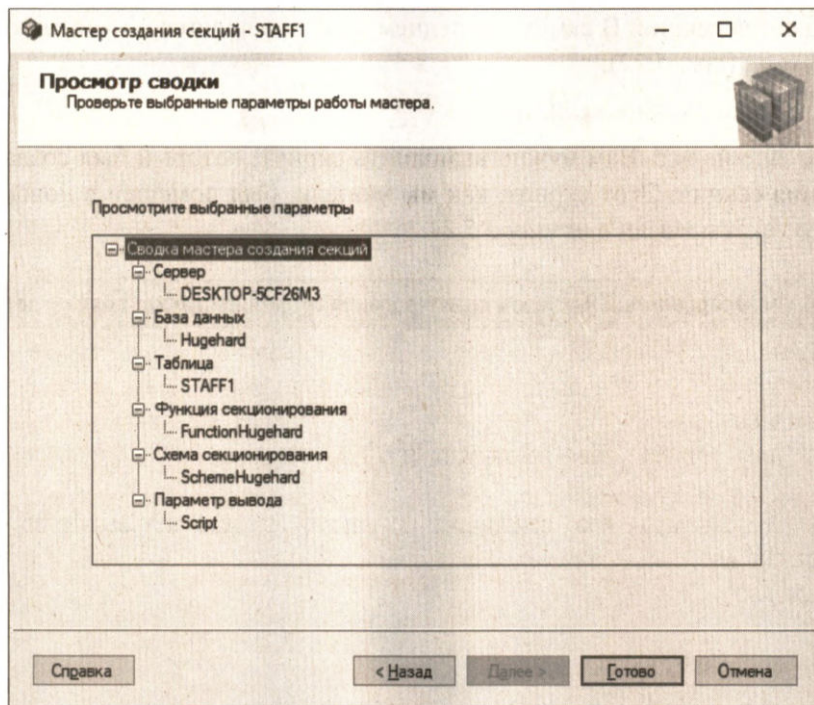


Рис. 5.21. Итоговые данные по выполненной работе

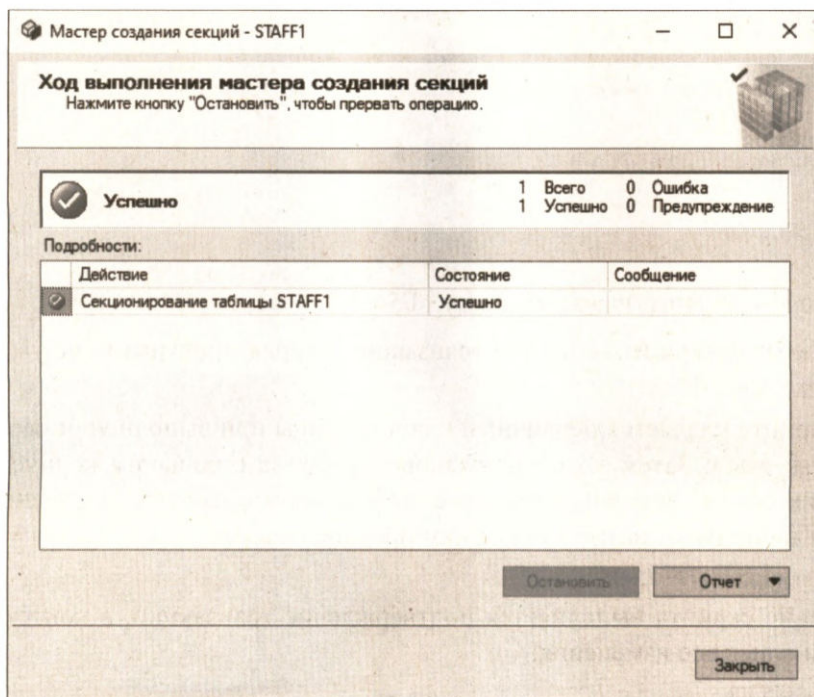


Рис. 5.22. Завершение секционирования таблицы

Далее создаются секции. В самом последнем окне сообщается об успешном выполнении действий (рис. 5.22).

Щелкните мышью по кнопке **Заккрыть**.

Однако это еще не все. Вам нужно выполнить скрипт, который был создан Мастером создания секций. Этот скрипт, как мы указали, был помещен 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>.

Первый оператор скрипта — запуск 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 во втором операторе последней строки изменения таблицы записано нужное нам предложение ON:

```
ON [SchemeHugehard] ([REGION])
```

В завершение скрипта выполняется подтверждение транзакции, в контексте которой выполнялись все изменения:

```
COMMIT TRANSACTION
```

О транзакциях мы с вами будем говорить в дальнейшем.

Честно скажу, я утомился, создавая секционированную таблицу в диалоговых средствах Management Studio. Лишний раз убедился, что подобные действия намного удобнее выполнять средствами 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 давайте создадим новую таблицу и сформируем средства секционирования "с нуля", т. е. создав схему и функцию секционирования с помощью диалоговых средств.

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 (*пример 5.25*).

Пример 5.25. Создание таблицы STAFF2 средствами Transact-SQL

```
USE Hugehard
GO
CREATE TABLE dbo.STAFF2
( 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_STAFF2 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),
CONSTRAINT CH_STAFF2
CHECK ((REGION='5' OR
       REGION='4' OR
       REGION='3' OR
       REGION='2' OR
       REGION='1')));
GO
```

Для создания средств секционирования в окне **Обозреватель объектов** щелкните правой кнопкой мыши по имени вновь созданной таблицы и выберите в контекстном меню элементы **Хранилище | Создать секцию**. Появится начальное окно Мастера создания секционирования. Его мы уже видели ранее (см. *рис. 5.16*). После щелчка по кнопке **Далее** появляется окно выбора разделяющего ключа (столбца деления) (*рис. 5.23*).

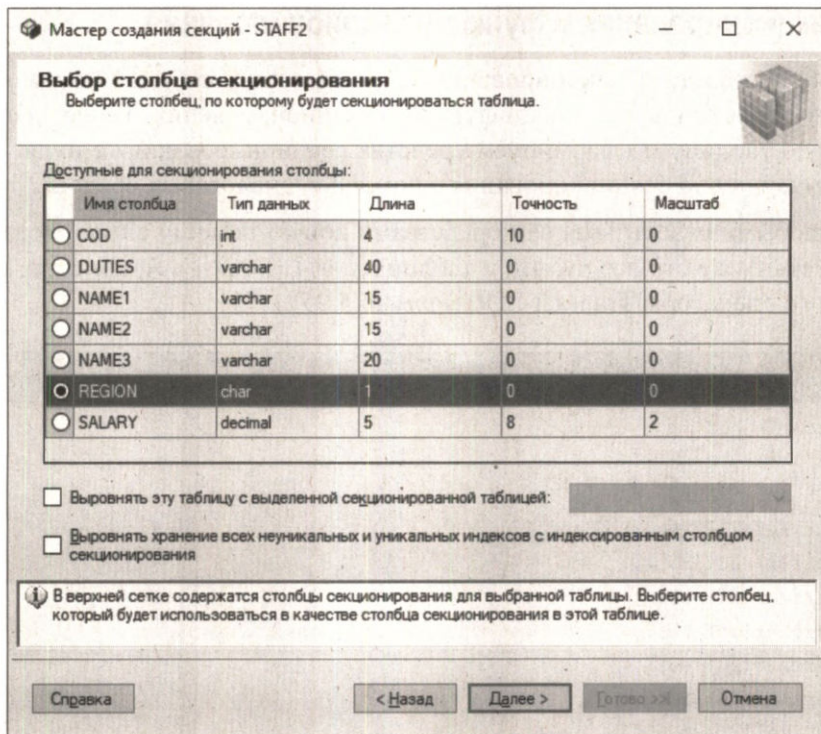


Рис. 5.23. Выбор столбца деления

Отмечаем столбец **REGION** и щелкаем по кнопке **Далее**. В следующем окне (*рис. 5.24*) указываем, что будем создавать новую функцию секционирования, и вводим в поле **Новая функция секционирования** имя функции — **FunctionHugehard2**.

После щелчка по кнопке **Далее** в следующем окне (*рис. 5.25*) мы указываем, что будем создавать новую схему секционирования, щелкнув мышью по радиокнопке **Новая схема секционирования**.

Вводим в этом поле имя схемы **SchemeHugehard2**.

Щелкаем по кнопке **Далее**.

В следующем окне (*рис. 5.26*) нужно создать условия секционирования таблицы.

Здесь в столбце **Файловая группа** нужно поочередно для каждой строки из выпадающего списка выбрать имена файловых групп: **America**, **Europe**, **Asia**, **Africa**, **Australia** и **Antarctica**.

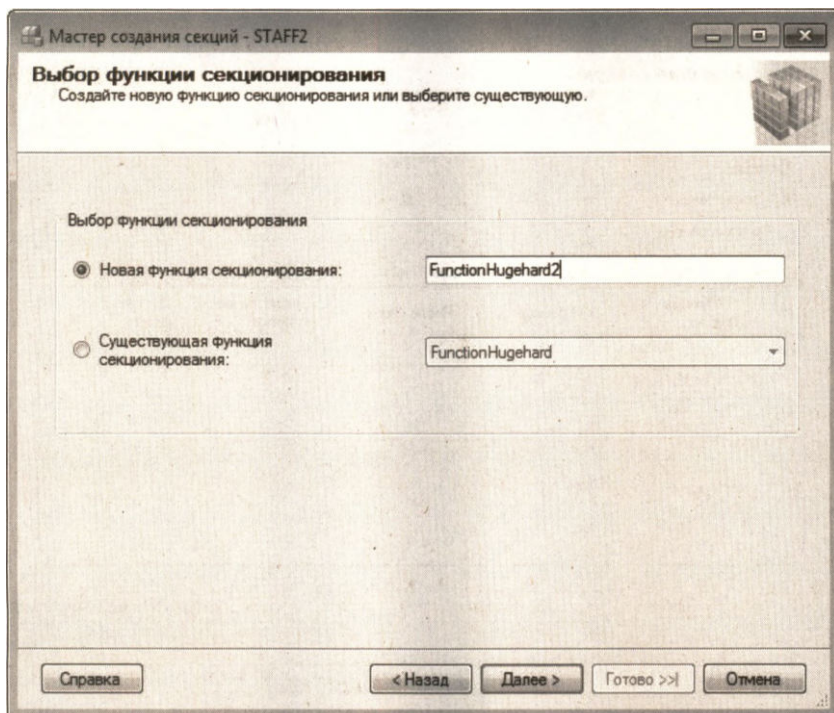


Рис. 5.24. Задание новой функции секционирования

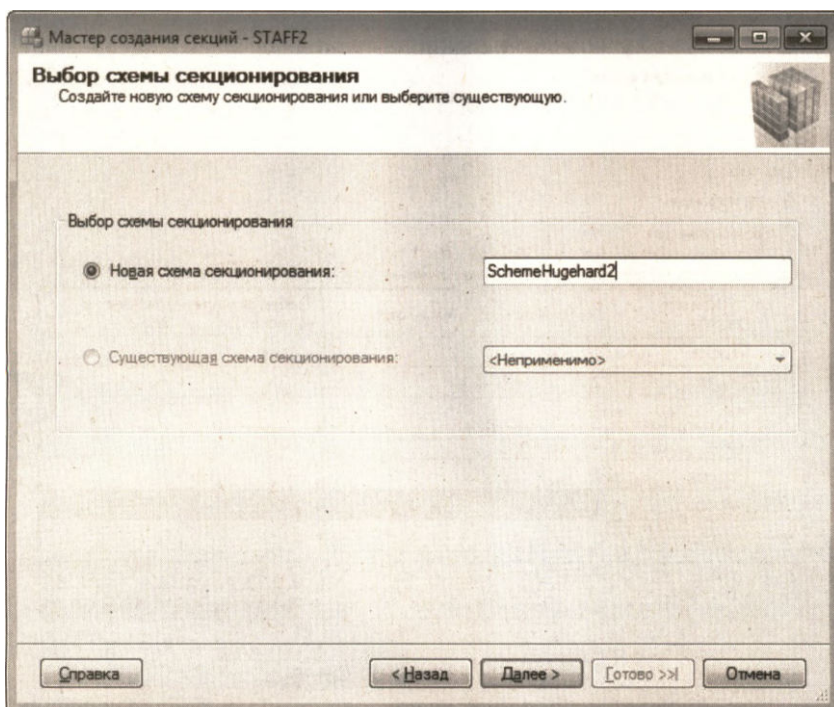


Рис. 5.25. Задание новой схемы секционирования

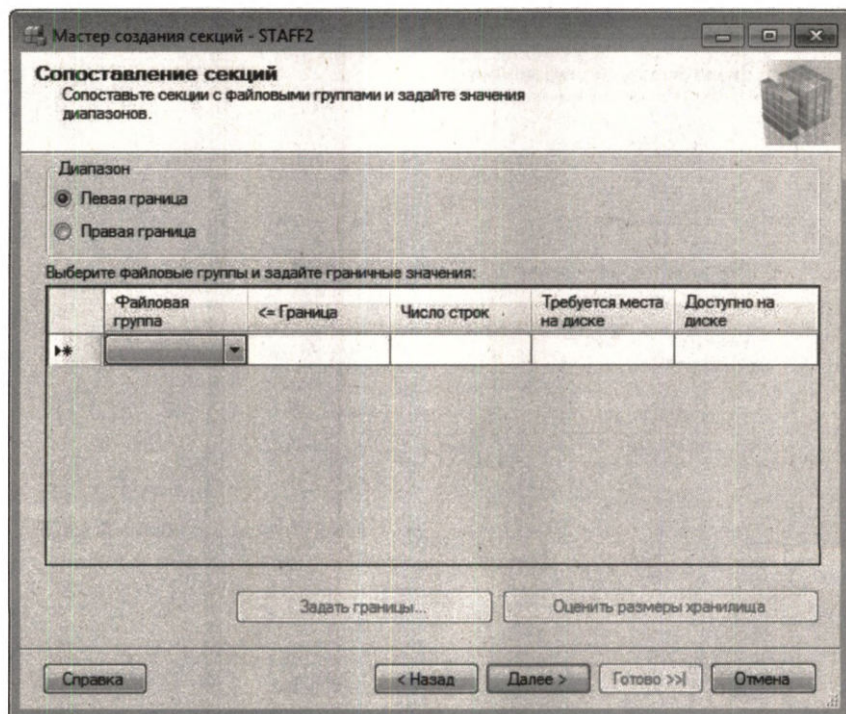


Рис. 5.26. Окно задания условий секционирования таблицы

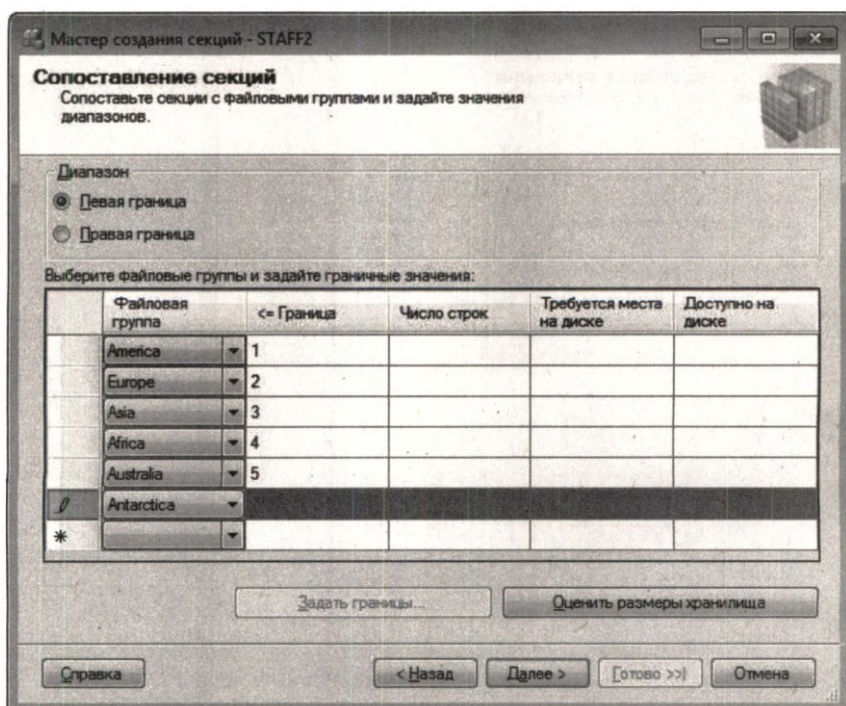


Рис. 5.27. Заданные условия секционирования таблицы

В следующем столбце задания границ значений \leq **Граница** 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. 5.27*.

Для последней строки, **Antarctica**, значение границы не указывается. Это будут все значения, которые превышают предыдущее значение, 5.

Щелкаем по кнопке **Далее** и в следующем окне (*рис. 5.28*) отмечаем, что требуется создание скрипта в новом окне запросов (отмечаем радиокнопку **Вывести скрипт в новое окно запроса**).

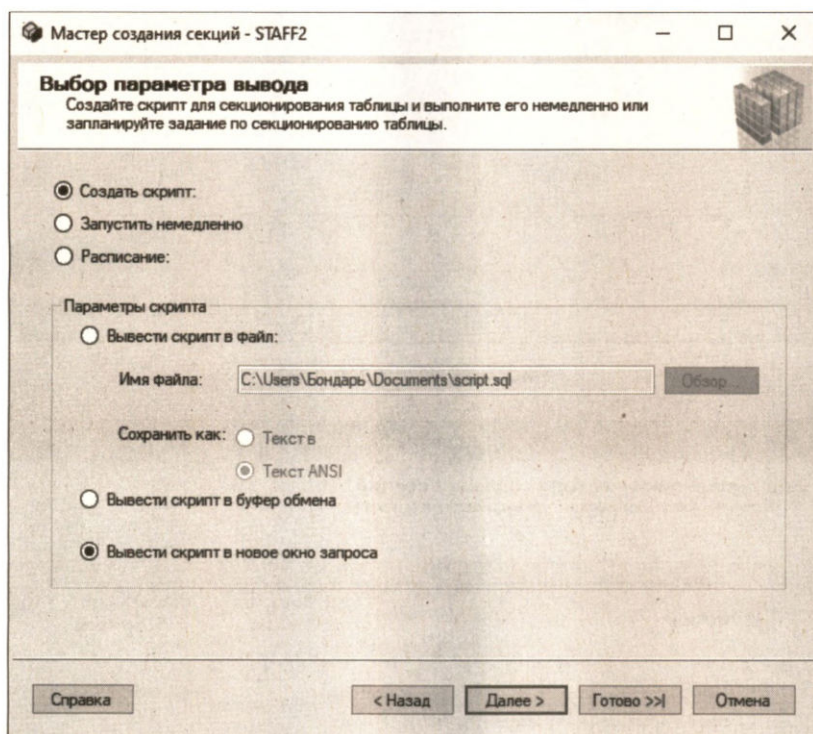


Рис. 5.28. Указание, что необходимо создание скрипта

После щелчка по кнопке **Далее** мы увидим итоговое окно (*рис. 5.29*), в котором нужно только щелкнуть мышью по кнопке **Готово**.

Последним будет окно завершения работы Мастера с перечнем создаваемых объектов и с указанием успешности выполнения необходимых действий (*рис. 5.30*).

Закройте это последнее окно, щелкнув мышью по кнопке **Заккрыть**.

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.

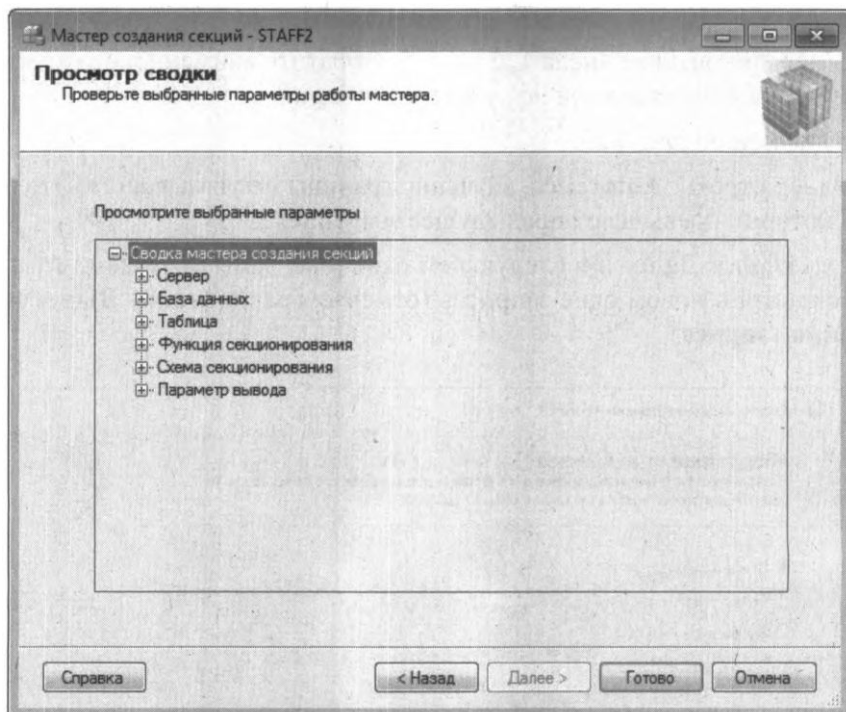


Рис. 5.29. Итоговое окно

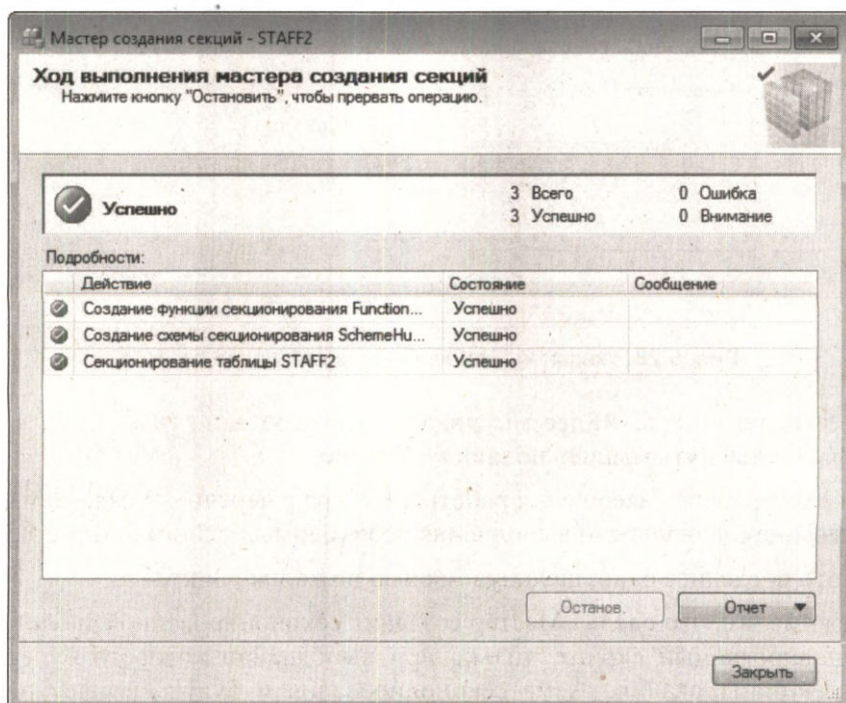


Рис. 5.30. Завершение работы Мастера

Скрипт показан в *примере 5.26*.

Пример 5.26. 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 **Обозреватель объектов** можно увидеть, что в базе данных появилась новая функция секционирования и новая схема секционирования. Чтобы их действительно увидеть, необходимо обновить список. Для этого нужно по имени базы данных **Hugehard** щелкнуть правой кнопкой мыши и в контекстном меню выбрать элемент **Обновить**.

5.5. Отображение состояния секционированных таблиц

На текущий момент у нас в БД 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 элементы **Отчеты | Стандартный отчет | Использование дисковой памяти секциями**.

В результате будет создан отчет, как показано на *рис. 5.31*.

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 число записей.

Имя таблицы	Число записей	Зарезервировано (КБ)	Используется (КБ)
<input type="checkbox"/> dbo.STAFF	120	360	80
<input type="checkbox"/> Индекс (PK_STAFF)	120	360	80
Номер секции	Число записей	Зарезервировано (КБ)	Используется (КБ)
1	20	72	16
2	20	72	16
3	20	72	16
4	20	72	16
5	40	72	16
<input type="checkbox"/> dbo.STAFF1	0	0	0
<input type="checkbox"/> Индекс (PK_STAFF1)	0	0	0
Номер секции	Число записей	Зарезервировано (КБ)	Используется (КБ)
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
<input type="checkbox"/> dbo.STAFF2	0	0	0
<input type="checkbox"/> Индекс (PK_STAFF2)	0	0	0
Номер секции	Число записей	Зарезервировано (КБ)	Используется (КБ)
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0

Рис. 5.31. Распределение данных по файловым группам секционированной таблицы

5.6. Файловые потоки

Прежде чем приступить к исследованию возможностей файловых потоков, следует проверить, поддерживает ли ваша конфигурация сервера базы данных такие средства. При установке системы эта поддержка была задана по умолчанию. Тем не менее, на всякий случай выполните следующую проверку и при необходимости внесите нужные коррективы.

Запустите на выполнение программу SQL Server Configuration Manager (Диспетчер конфигурации SQL Server) (рис. 5.32).

В правой части окна щелкните правой кнопкой мыши по имени сервера. В моем случае это строка **SQL Server (MSSQLSERVER)**. В контекстном меню выберите элемент **Свойства**. Появится диалоговое окно свойств сервера БД. Выберите вкладку **FILESTREAM** (рис. 5.33).

Здесь нужно отметить флажок допустимости файловых потоков в обычных операторах Transact-SQL: **Разрешить FILESTREAM при доступе через Transact-SQL**. 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. Иными словами, при изменении дан-

ных файлового потока будут корректироваться и данные в файловой системе. При удалении строк таблицы данные файлового потока также будут физически удаляться и из файлов файловой системы.

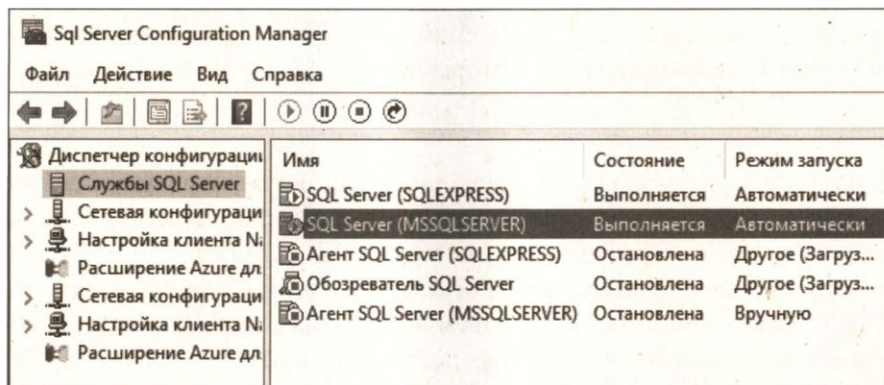


Рис. 5.32. Главное окно программы SQL Server Configuration Manager

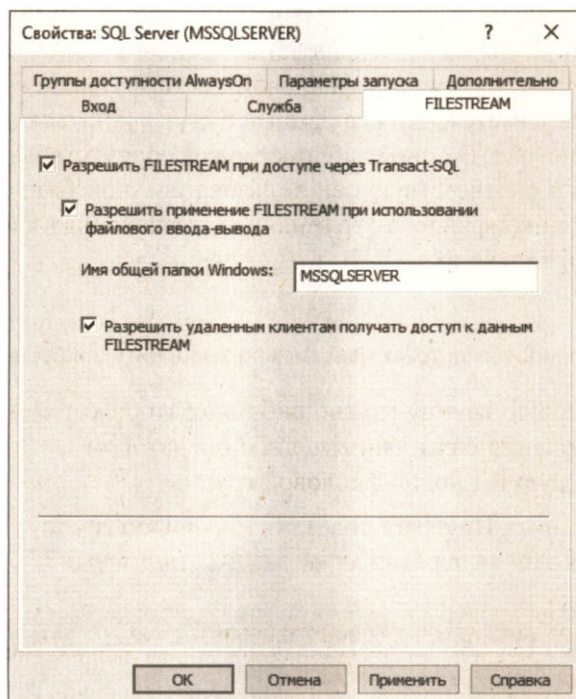


Рис. 5.33. Вкладка FILESTREAM свойств сервера

Если нужно, чтобы к файлам потока осуществлялся доступ средствами операционной системы, отметьте флажок **Разрешить применение FILESTREAM при использовании файлового ввода-вывода**. При этом в поле **Имя общей папки Windows** нужно ввести имя сервера БД: **MSSQLSERVER**.

Если вы хотите, чтобы с файловыми потоками могли работать и удаленные клиенты, то следует отметить и флажок **Разрешить удаленным клиентам получать доступ к данным FILESTREAM**.

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:

```
Sqlcmd 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 БД называется обычно двоичным большим объектом **BLOB** (Binary Large Object) или **LOB**. Эти данные могут храниться в самой БД. Тогда максимальный объем памяти, который они могут занимать, ограничивается размером 2 Гб. Размер достаточно большой, однако существуют приложения, требующие еще большего размера.

В SQL Server есть средства, позволяющие увеличить размер памяти, используемой для хранения таких 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 БД, 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.

Создать такую файловую группу можно либо оператором **CREATE DATABASE** в случае первоначального создания базы данных, либо оператором **ALTER DATABASE** для помещения в существующую БД новой файловой группы.

Давайте для базы данных Hугехард создадим файловую группу для файловых потоков. Используем для этого оператор **ALTER DATABASE** (*пример 5.27*).

Пример 5.27. Добавление в базу данных новой файловой группы

```
USE master;  
GO  
ALTER DATABASE Hугеhard  
    ADD FILEGROUP GroupStream CONTAINS FILESTREAM;  
GO  
ALTER DATABASE Hугеhard  
    ADD FILE  
        (NAME = GroupStream,
```

```

FILENAME = 'd:\Hugehard\GroupStream')
TO FILEGROUP GroupStream;
GO

```

За одно выполнение оператора ALTER DATABASE можно осуществить 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 будут созданы файл filestream.hdr и каталог \$FSLOG.

Чтобы просмотреть список файловых групп базы данных, щелкните правой кнопкой мыши по имени БД Hugehard в окне **Обозреватель объектов**, выберите в контекстном меню элемент **Свойства** и в появившемся окне в левом верхнем углу выберите вкладку **Файловые группы** (рис. 5.34).

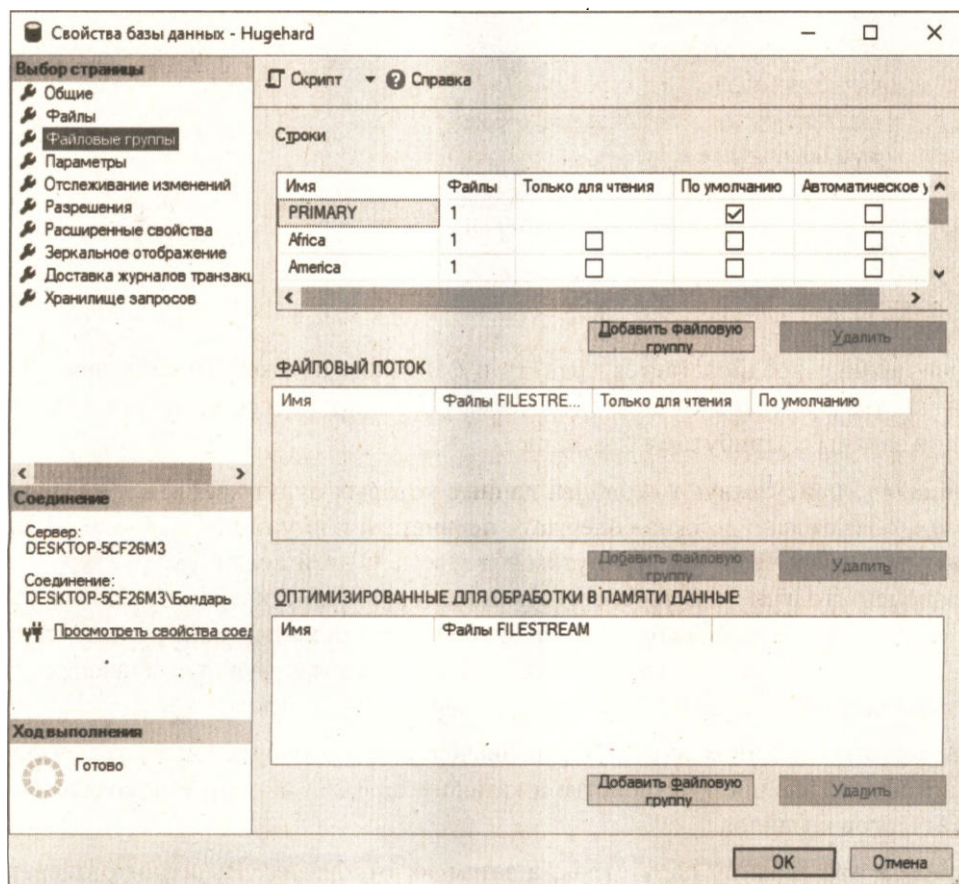


Рис. 5.34. Файловые группы базы данных Hugehard

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 5.28*.

Пример 5.28. 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, /* Карта страны */
  DESCR VARBINARY(MAX)
    FILESTREAM, /* Дополнительное описание */
  CONSTRAINT PK_REFCTR PRIMARY KEY (CODCTR)
);
GO
```

В этой таблице предполагается хранить в файловом потоке значения двух столбцов — MAP (карта страны) и DESCR (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 5.29*. Здесь в таблицу добавляются две строки.

Пример 5.29. Внесение данных в таблицу, использующую файловые потоки

```

USE Hugehard;
INSERT INTO REFCTR (CODCTR, NAME, DESCR)
VALUES ('RUS', 'Россия',
        CAST('Произвольное описание' AS VARBINARY(MAX)));
INSERT INTO REFCTR (CODCTR, NAME, DESCR)
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 явно преобразовать строку 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 5.30.

Пример 5.30. Отображение данных таблицы, использующей файловые потоки

```

USE Hugehard;
SELECT CODCTR AS 'Код',
       NAME AS 'Название',
       DESCR AS 'Описание'
FROM REFCTR;
GO

```

Результат будет не очень наглядным. Двоичные данные отображаются именно как двоичные:

Код	Название	Описание
RUS	Россия	0xCFF0EEE8E7E2EEEBFCDEEE520EEFE8F1E0EDE8E5
USA	United States	0xD2EEE6E520EDE5EAEEF2EEF0EEE520EEFE8F1E0EDE8E5

Немного изменим оператор выборки данных, добавив отображение идентификатора `ID` и выполнив преобразование двоичных данных к строковому типу данных переменной длины (пример 5.31).

Пример 5.31. 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

Несмотря на то, что данные описания хранятся в отдельном файле, мы можем с ними работать точно так же, как и с любыми данными, хранящимися в самой БД.

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, т.е. если на таблицу ссылается хранимая процедура, представление (VIEW) или когда на первичный или уникальный ключ таблицы ссылается внешний ключ другой таблицы.

Для выполнения всех последующих действий с базой данных в этой главе пересоздайте и заполните данными БД BestDatabase, используя четыре скрипта с сайта издательства (см. приложение 6).

5.7.1. Определение зависимостей таблицы

Проще всего определить зависимости таблицы в Management Studio.

В окне **Обозреватель объектов** раскройте папку **Базы данных, BestDatabase** и папку **Таблицы**. Щелкните правой кнопкой мыши по имени таблицы **dbo.REFCTR**. Выберите в контекстном меню элемент **Просмотреть зависимости**. Появится окно, в котором при отмеченной радиокнопке **Объекты, зависящие от [REFCTR]** показаны все объекты базы данных, которые так или иначе ссылаются на таблицу REFCTR (рис. 5.35).

Если раскрыть все элементы дерева, то можно увидеть две таблицы, которые напрямую зависят от справочника стран.

Чтобы просмотреть зависимости таблицы районов, щелкните правой кнопкой мыши по имени таблицы **dbo.REFAREA**. Выберите в контекстном меню элемент **Просмотреть зависимости**. Если отметить вторую радиокнопку **Объекты, от которых зависит [REFAREA]**, то можно увидеть список объектов, от которых зависит эта таблица (рис. 5.36).

Здесь видно, что таблица районов зависит от таблицы регионов, которая в свою очередь зависит от таблицы стран.

Другие средства определения зависимостей в базе данных не дают хорошего результата. 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 удобном виде многие характеристики таблиц.

Например, системная функция `sp_helpconstraint` позволяет отобразить все ограничения таблицы. Следующий пакет (пример 5.32) дает возможность отобразить все ограничения таблицы `REFCTR` базы данных `BestDatabase`.

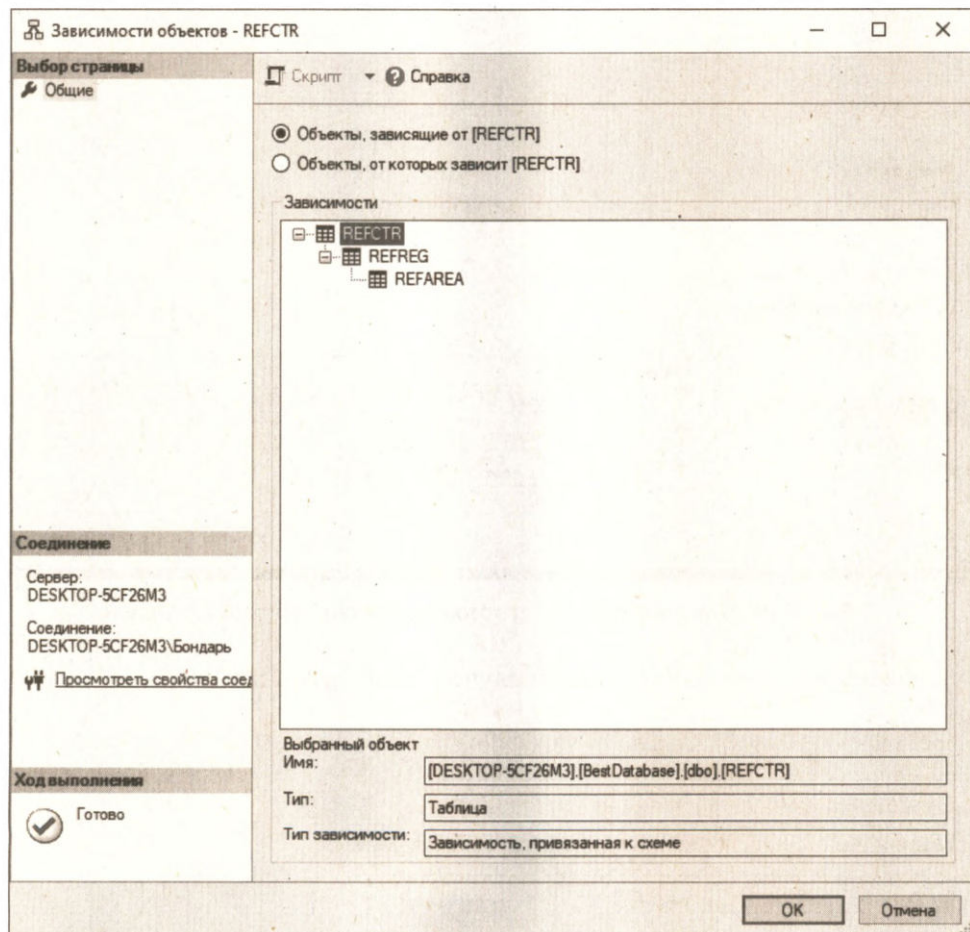


Рис. 5.35. Список объектов, зависящих от таблицы `REFCTR`

Пример 5.32. Отображение ограничений таблицы `REFCTR`

```
USE BestDatabase;  
GO  
EXEC sp_helpconstraint 'dbo.REFCTR';  
GO
```

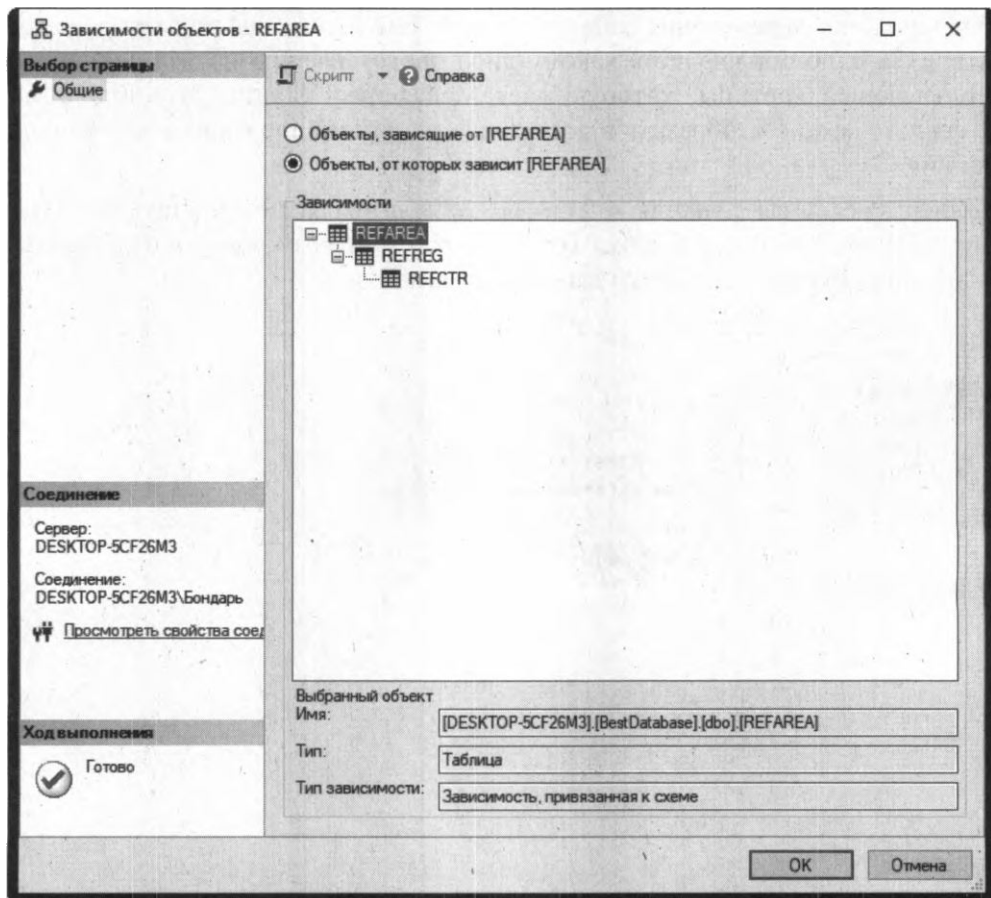


Рис. 5.36. Список объектов, от которых зависит таблица REFAREA

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.REFREG: FK_REFREG

Здесь отображаются:

- ◆ имя исходной таблицы — dbo.REFCTR;
- ◆ сведения о первичном ключе таблицы — PRIMARY KEY (clustered) PK_REFCTR CODCTR;

♦ внешний ключ таблицы, ссылающейся на этот первичный ключ, — `BestDatabase.dbo.REFREG: FK_REFREG`.

Можете отобразить ограничения таблицы `PEOPLE`. Там будут показаны 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` приведен в листинге 5.19.

Листинг 5.19. Синтаксис оператора удаления таблицы `DROP TABLE`

```
DROP TABLE [[<имя базы данных>.]<имя схемы>.]<имя таблицы>  
[, [[<имя базы данных>.]<имя схемы>.]<имя таблицы>]...;
```

Как и в случае оператора создания таблицы здесь также можно задать имя БД и имя схемы, отделяя их символом точки. Если в операторе не указано имя базы данных, то таблица удаляется из текущей БД, которая была установлена в последнем операторе `USE`. Если не указано имя схемы, то таблица удаляется из схемы БД по умолчанию — обычно это схема `dbo`.

В одном операторе можно удалить несколько таблиц, перечислив их имена через запятую.

5.7.3. Удаление таблицы диалоговыми средствами Manager Studio

Чтобы удалить таблицу с использованием диалоговых средств Management Studio, нужно в окне **Обозреватель объектов** по имени удаляемой таблицы щелкнуть правой кнопкой мыши и выбрать в контекстном меню элемент **Удалить**. Появится окно удаления объекта, как показано на *рис. 5.37*.

Чтобы просмотреть зависимости удаляемой таблицы, нужно щелкнуть по кнопке **Показать зависимости**.

Если вы захотите удалить таблицу, для которой существуют внешние зависимости, т. е. в БД присутствуют объекты, которые зависят от данной таблицы, то вы получите сообщение об ошибке, как показано на *рис. 5.38*.

Чтобы просмотреть сообщение о причинах невозможности удаления таблицы, щелкните мышью по гиперссылке в столбце **Сообщение**. Появится сообщение, показанное на *рис. 5.39*.

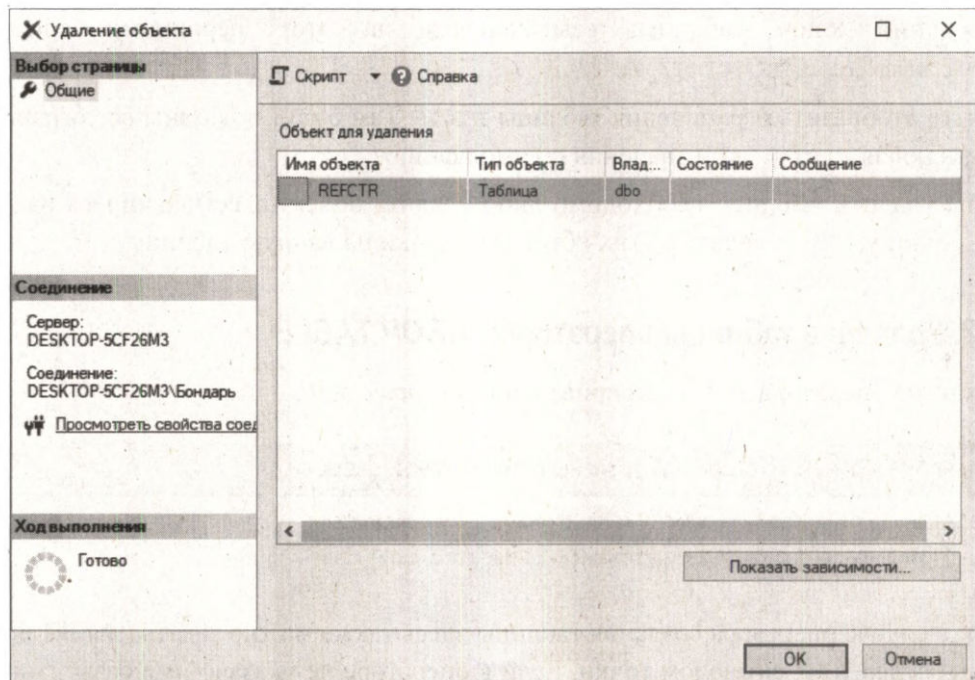


Рис. 5.37. Окно удаления объекта базы данных

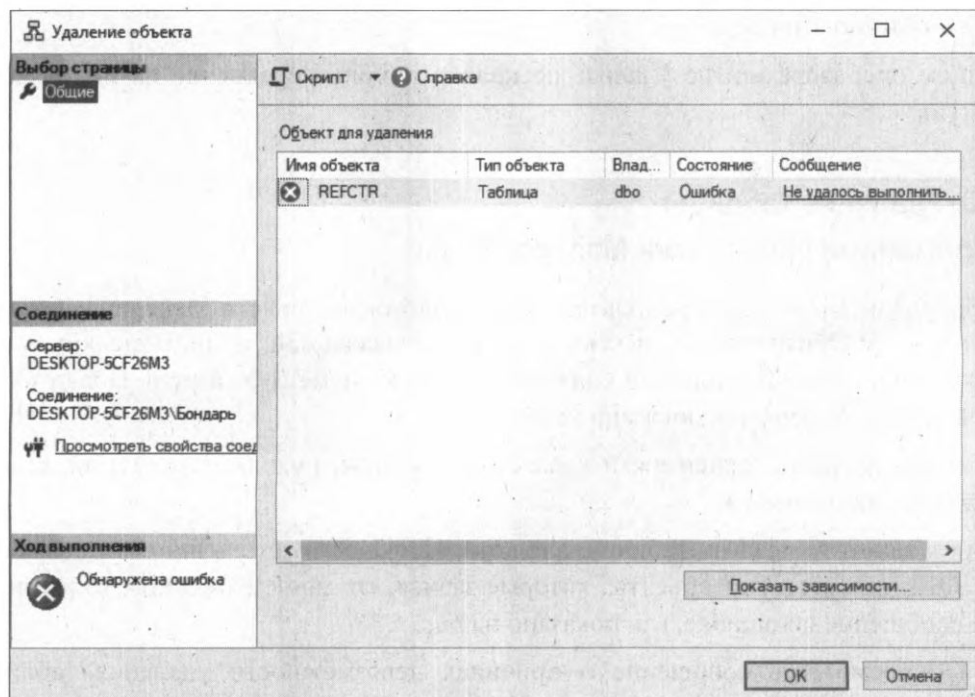


Рис. 5.38. Сообщение об ошибке при попытке удаления объекта БД, на который имеются ссылки других объектов БД

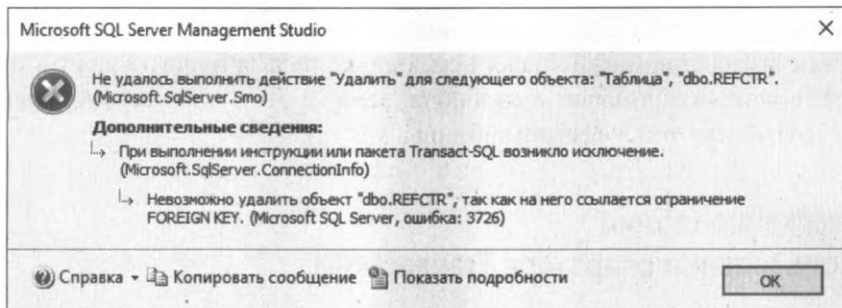


Рис. 5.39. Сообщение о причинах невозможности удаления таблицы

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 автоматически изменяется имя родительской таблицы.
- ◆ Удалить столбец таблицы. Нельзя удалить столбец, который
 - входит в состав такого ограничения таблицы, как первичный, уникальный, внешний ключ, `CHECK`;
 - служит 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 число 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 отзеркаливается 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. Для ограничения CHECK нет проблем с изменением или удалением, однако 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 приведен в листинге 5.20.

Листинг 5.20. 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 не показаны в *листинге 5.20*. В частности, отсутствуют подробные описания ограничений столбца и таблицы. При необходимости их можно посмотреть в соответствующих листингах в начале этой главы.

5.8.1.1. Изменение имени таблицы

Как и при создании или удалении таблицы в операторе `ALTER TABLE` 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 БД — в той базе, которая была установлена в последнем операторе `USE`. Если не указано и имя схемы, то изменяемая таблица выбирается в схеме БД 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 число 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 и не 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.

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":

Сообщение 5074, уровень 16, состояние 1, строка 3
объект "PK_REFCTR" зависит от столбец "CODCTR".

Сообщение 5074, уровень 16, состояние 1, строка 3
объект "FK_REFREG" зависит от столбец "CODCTR".

Сообщение 4922, уровень 16, состояние 9, строка 3

Ошибка ALTER TABLE ALTER COLUMN CODCTR, так как один или несколько объектов обращаются к данному столбцу.

Другой пример. Попытка изменить в сторону увеличения характеристик 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:

Сообщение 5074, уровень 16, состояние 1, строка 3
столбец "NET_SALARY" зависит от столбец "SALARY".

Сообщение 4922, уровень 16, состояние 9, строка 3

Ошибка ALTER TABLE ALTER COLUMN SALARY, так как один или несколько объектов обращаются к данному столбцу.

ЗАМЕЧАНИЕ

Иногда качество сообщений вызывает удивление у знатоков русского языка, например последнее сообщение. Не принимайте это близко к сердцу.

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 работал тогда с другой СУБД, и такое изменение у меня не прошло.

Еще пример:

```
USE BestDatabase;
GO
ALTER TABLE REFCTR
    ALTER COLUMN FULLNAME VARCHAR(70);
GO
```

Здесь увеличивается число символов в столбце FULLNAME со строковым типом данных переменной длины с 65 до 70.

Однако попытка уменьшить размер строкового данных, сделать его меньше, чем размер данных, уже существующих в таблице, даст ошибку. Например, при выполнении такого пакета

```
USE BestDatabase;
GO
ALTER TABLE REFCTR
    ALTER COLUMN FULLNAME VARCHAR(60);
GO
```

вы получите сообщение об ошибке, в котором говорится, что возможно усечение строковых или двоичных данных.

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 числа знаков строковых типов данных в сторону увеличения, если нужные вам тексты не помещаются в существующие заданные размеры, или 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 поменять 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 можно изменить порядок сортировки столбца FULLNAME таблицы REFCTR, выполнив 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 `REFSTR`. 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, нужно учитывать только тот факт, что число столбцов в таблице не должно превышать 1024 (30 000 при наличии 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.

Добавить обычный или вычисляемый столбец можно 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 описании 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, данные которого хранятся в файловом потоке, эти данные не будут удалены с внешнего носителя до перезапуска сервера БД.

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

И в SQL Server версии 2022 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

Диалоговые средства для изменения таблиц намного удобнее в использовании, чем оператор `ALTER TABLE`.

5.8.2.1. Изменение имени таблицы

В окне **Обозреватель объектов** нужно раскрыть базу данных, затем папку **Таблицы**, щелкнуть правой кнопкой мыши по имени таблицы и в контекстном меню выбрать элемент **Переименовать**. Имя таблицы в списке 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 БД. 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 **Проект**. В основной части окна Management Studio появится вкладка проектирования таблицы, содержащая список столбцов таблицы и некоторых их характеристик. В нижней части главного окна располагается вкладка **Свойства столбца**, в которой описаны характеристики (свойства) выбранного столбца таблицы. В главном меню программы появится элемент **Конструктор таблиц**, 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. 5.40* показан список столбцов в режиме **Проект** таблицы `REFCTR` нашей базы данных `BestDatabase`.


	Имя столбца	Тип данных	Разрешить значения NULL
	CODCTR	char(3)	<input type="checkbox"/>
	NAME	varchar(60)	<input checked="" type="checkbox"/>
	FULLNAME	varchar(65)	<input checked="" type="checkbox"/>
	CAPITAL	varchar(30)	<input checked="" type="checkbox"/>
	MAP	varbinary(MAX)	<input checked="" type="checkbox"/>
	DESCR	varbinary(MAX)	<input checked="" type="checkbox"/>

Рис. 5.40. Список столбцов таблицы `REFCTR` во вкладке проектирования

Для каждого столбца указаны имя, тип данных и допустимость значения `NULL`. В самом левом столбце значок указывает, входит ли этот столбец таблицы в состав первичного ключа.

На *рис. 5.41* показаны характеристики столбца `CODCTR`, отображаемые во вкладке **Свойства столбца** в нижней части окна проектирования таблицы.

▼ (Общие)	
(Имя)	CODCTR
Длина	3
Значение по умолчанию или привязка	
Разрешить значения NULL	Нет
Тип данных	char
▼ Конструктор таблиц	
RowGuid	Нет
Детерминированный	Да
Имеет подписчик, отличный от подписчика SQL Server	Нет
Индексируемый	Да
Набор столбцов	Нет
Не для репликации	Нет
Описание	
Опубликован слиянием	Нет

Рис. 5.41. Свойства столбца CODCTR таблицы REFCTR

В этой вкладке мы можем видеть имя текущего (выделенного) столбца, допустимость значения NULL, тип его данных и отдельно размер строкового данного, порядок сортировки, значение по умолчанию, характеристики IDENTITY, если они установлены, формулу для получения значения вычисляемого столбца и ряд других.

5.8.2.3. Изменение типа данных

Изменить тип данных столбца можно во вкладке проектирования таблицы (см. рис. 5.40) или во вкладке просмотра свойств столбца (см. рис. 5.41).

Во вкладке проектирования таблицы нужно щелкнуть мышью по полю **Тип данных** у соответствующего столбца таблицы. Справа появится кнопка со стрелкой вниз. После щелчка по этой кнопке появится выпадающий список всех допустимых в этой БД системных и пользовательских типов данных.

На рис. 5.42 показан такой список для столбца NAME таблицы REFCTR.

Имя столбца	Тип данных
☺ CODCTR	char(3)
▶ NAME	varchar(60) ▼
FULLNAME	varbinary(50)
CAPITAL	varbinary(M...
MAP	varchar(60)
DESCR	varchar(MAX)
	xml
	D_INT:int
	D_CHAR30:v...
	BOOLEAN:bit ▼

Рис. 5.42. Типы данных для столбца NAME таблицы REFCTR

В этом списке нужно выбрать соответствующий тип данных. Здесь можно изменить размер строки. Давайте зададим 70. Это число нужно поместить в скобки по-

сле 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 во вкладке **Свойства столбца**. Щелкните мышью по строке **Тип данных**. Справа появится кнопка со стрелкой вниз. При щелчке мышью по этой кнопке появится выпадающий список допустимых системных и пользовательских типов данных (рис. 5.43).

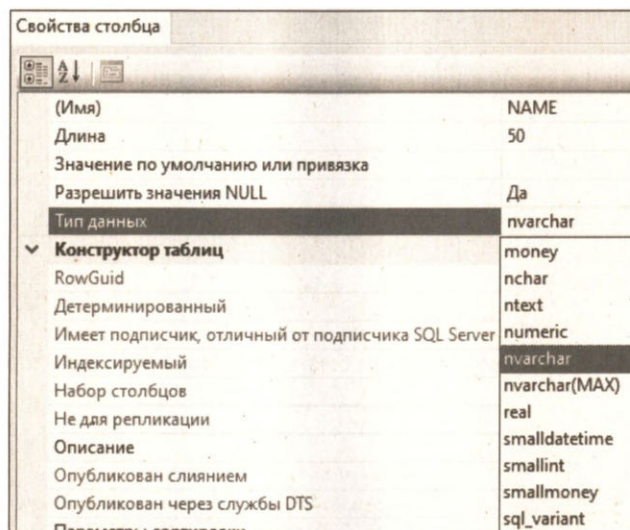


Рис. 5.43. Типы данных для столбца FULLNAME во вкладке **Свойства столбца**

Из списка выберите тип данных `nvarchar`. Следует установить для него размер 75. Для задания нового значения размера нужно чуть выше в списке свойств щелкнуть мышью по полю **Длина** и изменить заданный там размер поля на 75 и нажать клавишу <Enter>.

Все выполненные изменения характеристик столбца во вкладке **Свойства столбца** 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 и в контекстном меню выбрать строку **Сохранить REFCTR**. Однако вместо сохранения изменений мы получим следующее сообщение (рис. 5.44).

Здесь нужно щелкнуть мышью по кнопке **Отмена**, поскольку при существующих установках программы Management Studio (это значения по умолчанию) сохранить выполненные изменения невозможно.

Чтобы обеспечить возможность изменять характеристики столбцов таблицы, нужно установить соответствующее свойство. В главном меню программы щелкните мышью по элементу **Сервис** и выберите элемент **Параметры**.

Появится окно задания свойств (рис. 5.45).

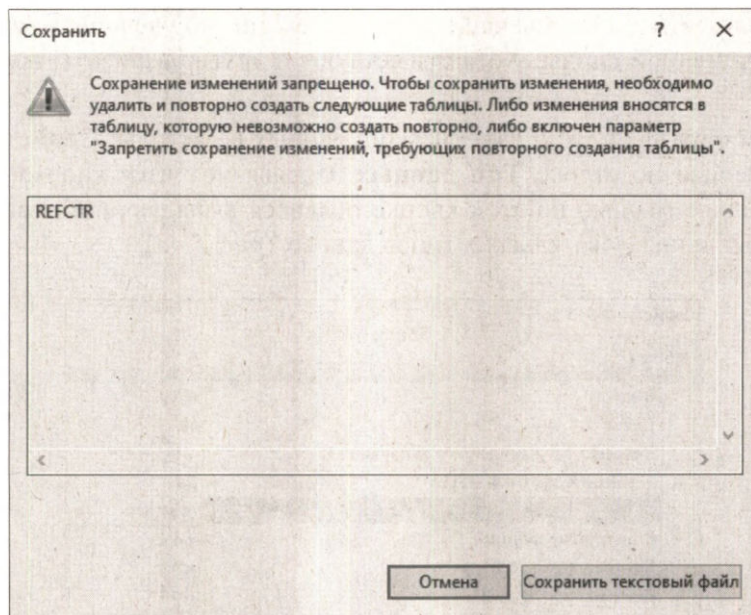


Рис. 5.44. Сообщение о невозможности сохранить изменения характеристик таблицы

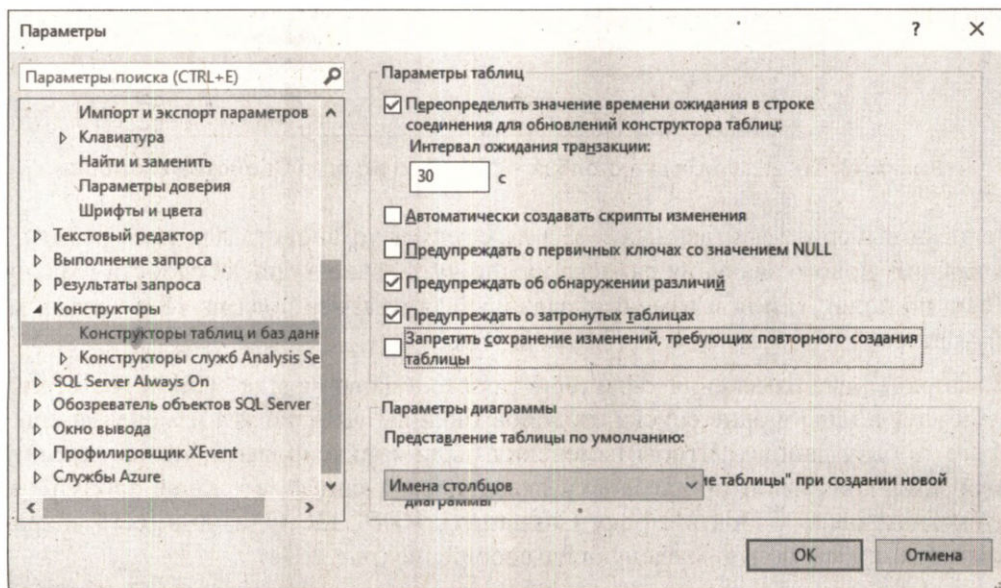


Рис. 5.45. Задание свойств программы

В левой части окна раскройте папку **Конструкторы**. В правой части окна найдите строку **Запретить сохранение изменений, требующих повторного создания таблицы** и снимите флажок. Щелкните мышью по кнопке **ОК**.

Теперь для реального сохранения изменений щелкните правой кнопкой мыши по заголовку вкладки окна списка столбцов таблицы и в контекстном меню выберите

строку **Сохранить REFCTR**. Появится окно (рис. 5.46), в котором сообщается о том, что будут сохранены (изменены) две таблицы в базе данных. Почему нужно вносить изменения в таблицу REFREG, я так и не понял.

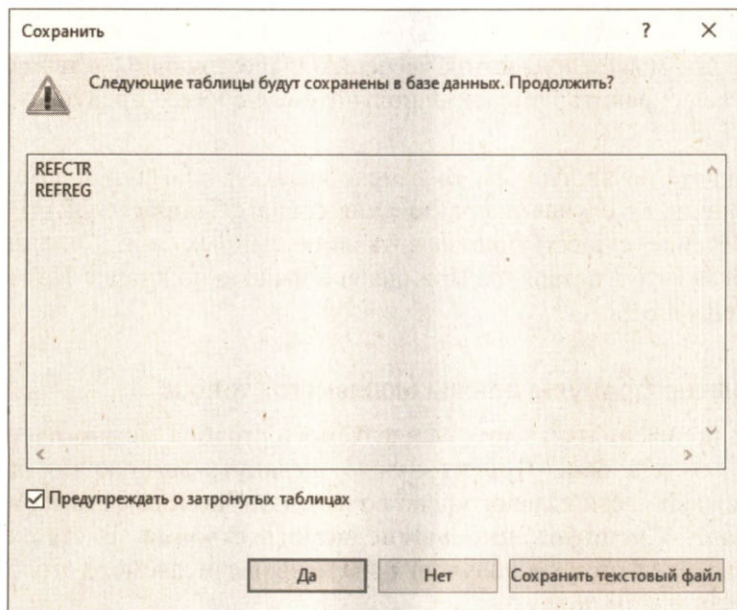


Рис. 5.46. Сообщение об изменяемых таблицах

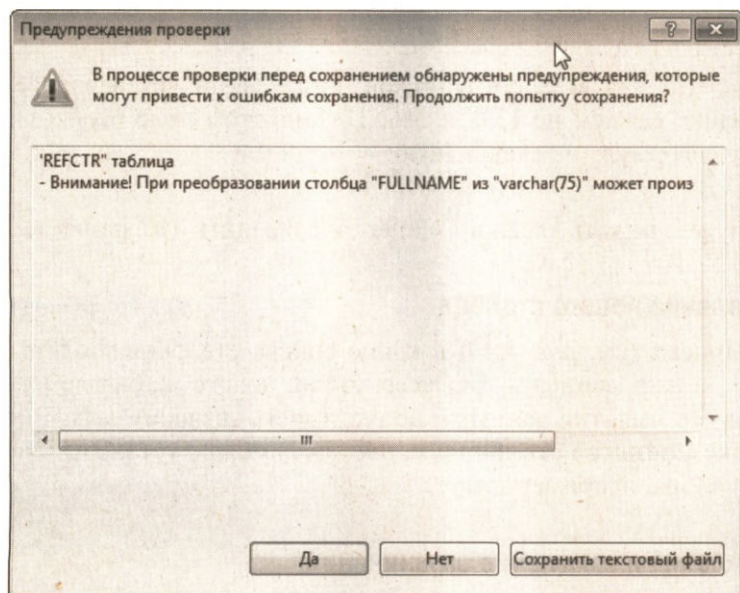


Рис. 5.47. Диагностическое сообщение о невозможности выполнить изменение размера

В этом окне щелкните мышью по кнопке **Да**. 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 **Заккрыть**.

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. 5.47).

Если вы щелкнете по кнопке **Да**, 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" символы справа в строках будут потеряны. При щелчке мышью по кнопке **Нет** изменения не будут сохранены в БД.

5.8.2.4. Изменение формулы для вычисляемого столбца

Формулу для получения значения вычисляемого столбца можно очень просто изменить. Для этого в окне **Проект** нужно щелкнуть мышью по вычисляемому столбцу. В нижней части главного окна во вкладке **Свойства столбца** необходимо раскрыть группу **Спецификация вычисляемого столбца**. В строке **(Формула)** 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** вычисляемый — 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 необходимо заменить существующую формулу на следующую:

```
(([SALARY] * (0.95))
```

После этого нужно нажать клавишу <Enter> и сохранить изменения таблицы.

5.8.2.5. Добавление нового столбца

Во вкладке **Проект** (см. рис. 5.40) 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 **Свойства столбца** (см. рис. 5.41) можно установить и другие свойства нового столбца, например **IDENTITY**.

5.8.2.6. 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 во вкладке **Проект**. Нужно мышью выделить столбец, который вы хотите сделать первичным ключом, и в главном меню выбрать элементы **Конструктор таблиц | Задать первичный ключ**. Можно также щелкнуть по имени столбца правой кнопкой мыши и в контекстном меню выбрать элемент **Задать первичный ключ**.

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.

Если же в таблице был первичный ключ, на который посредством внешних ключей ссылаются другие таблицы, то будет выдано сообщение, показанное на *рис. 5.48*.

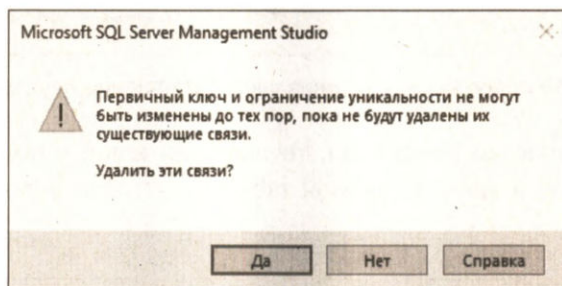


Рис. 5.48. Диагностическое сообщение о наличии связей других таблиц с первичным ключом таблицы

Сейчас я собираюсь изменить первичный ключ таблицы `REFACTIV`, которая была создана следующим оператором:

```
CREATE TABLE REFACTIV  
( COD      CHAR(4) NOT NULL,    /* Код вида деятельности */  
  NAME     VARCHAR(110),       /* Наименование вида деятельности */  
  CONSTRAINT PK_REFECTIV  
    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 вы щелкнете по кнопке **Да** и попытаетесь сохранить изменения, то появится следующее диалоговое окно, в котором спрашивается, следует ли вносить изменения в две таблицы базы данных (*рис. 5.49*). Здесь речь в первую очередь идет об удалении внешнего ключа в таблице `ORGACTIV`, который ссылается на первичный ключ изменяемой таблицы.

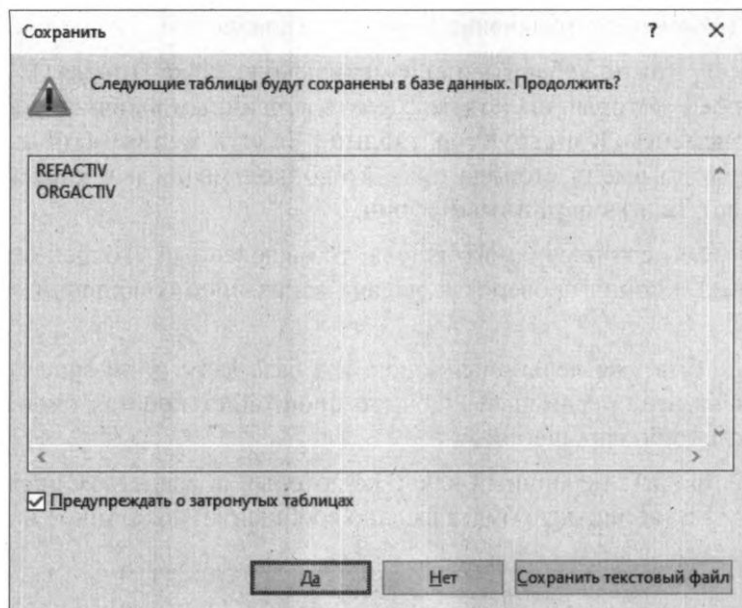


Рис. 5.49. Диалоговое окно, запрашивающее изменение двух других таблиц

Если щелкнуть мышью по кнопке **Да**, то внешний ключ у подчиненной таблицы **ORGACTIV** будет удален, а для изменяемой таблицы **REFACTIV** будет установлен новый первичный ключ.

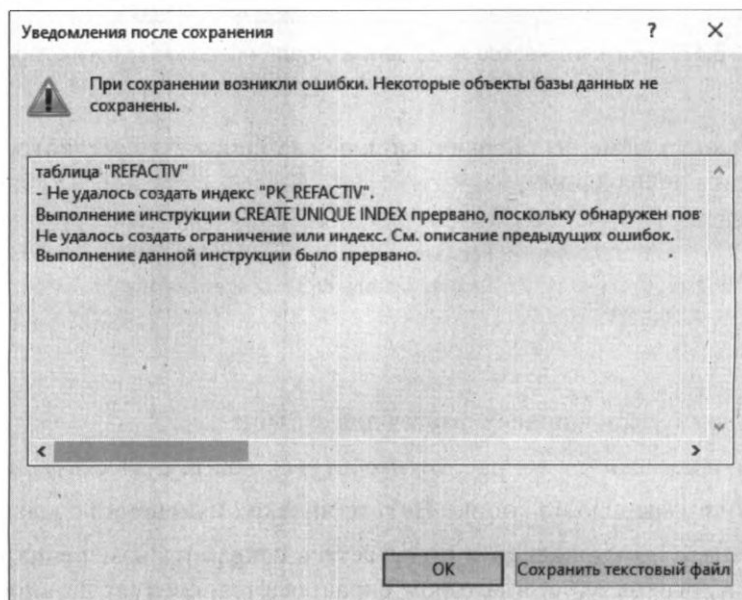


Рис. 5.50. Диалоговое окно, сообщающее о наличии дубликатов значений первичного ключа

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 **Да** будет отображено следующее диалоговое окно, сообщающее об ошибке (рис. 5.50).

В этом случае нужно только щелкнуть мышью по кнопке **ОК**. Первичный ключ у таблицы изменен не будет.

Первичный ключ состоит из нескольких столбцов таблицы

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 **Индексы и ключи**. 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 **Проект**. Выберите в меню элементы **Конструктор таблиц | Индексы и ключи** или щелкните правой кнопкой мыши по имени любого столбца таблицы и в контекстном меню выберите элемент **Индексы и ключи**. Появится окно **Индексы/Ключи** (рис. 5.51).

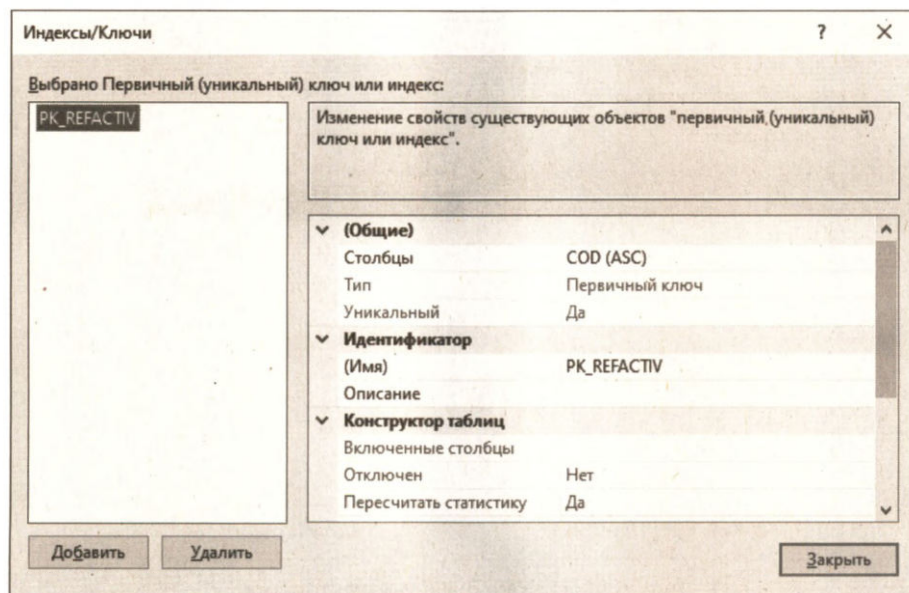



Рис. 5.51. Список ключей таблицы `REFACTIV`

Чтобы добавить в состав первичного ключа таблицы еще один столбец, выделите мышью строку **Столбцы**, щелкните в правой части этой строки по кнопке . Следующим будет окно, описывающее столбцы индекса первичного ключа данной таблицы (рис. 5.52).

Добавьте в список столбец `NAME`, выбрав его из выпадающего списка, и установите для него упорядоченность по возрастанию значений. Список станет следующим (рис. 5.53).

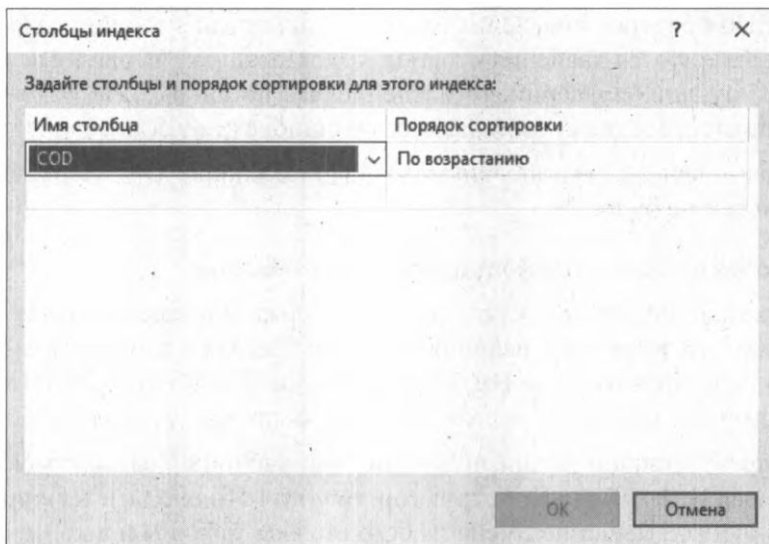


Рис. 5.52. Список столбцов первичного ключа таблицы REFACTIV

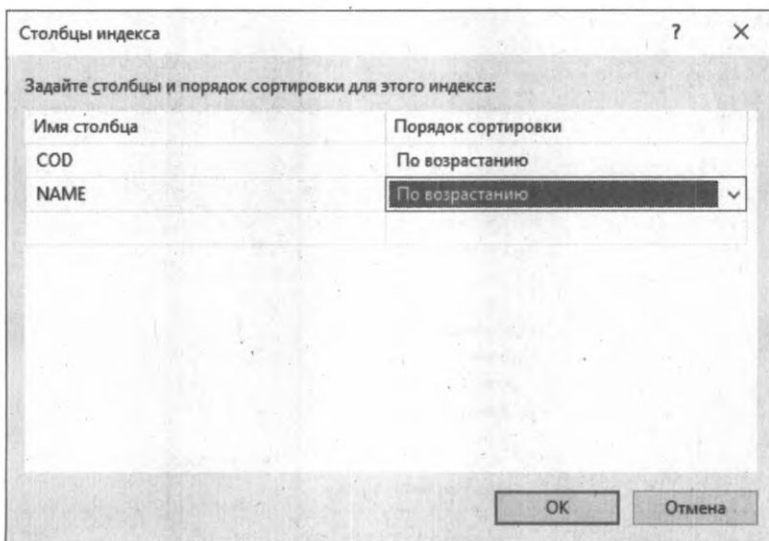


Рис. 5.53. Измененный список столбцов
первичного ключа таблицы REFACTIV

Щелкните мышью по кнопке **OK**. Появится предупреждающее окно, где сообщается, что прежде чем изменять ограничение первичного или уникального ключа нужно удалить существующее (рис. 5.54). Вас спрашивают, хотите ли вы удалить существующие отношения.

Щелкните мышью по кнопке **Да**. В окне **Индексы/Ключи** (см. рис. 5.51) щелкните по кнопке **Заккрыть**. В первичный ключ таблицы будет добавлен столбец **NAME**. Чтобы реально выполнить изменения, сохраните таблицу.

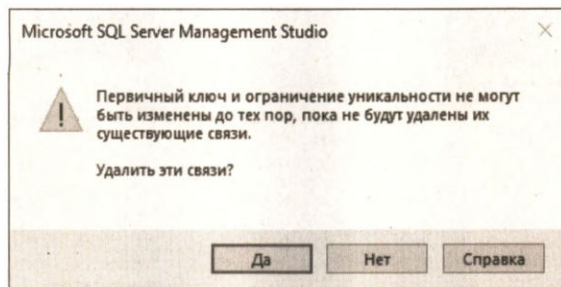


Рис. 5.54. Предупреждающее сообщение

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 нежелательно создавать большие по размеру первичные ключи. 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 перечисленные столбцы: `NAME3` (фамилия), `NAME1` (имя), `NAME2` (отчество) и `BIRTHDAY` (дата рождения).

Щелкните правой кнопкой мыши по имени таблицы `PEOPLE` в **Обозревателе объектов** и в контекстном меню выберите элемент **Проект**. В главном окне появится вкладка, содержащая список столбцов таблицы.

Выберите в главном меню элементы **Конструктор таблиц | Индексы и ключи** или щелкните правой кнопкой мыши по любому столбцу в списке столбцов таблицы и выберите в контекстном меню элемент **Индексы и ключи**.

Появится окно просмотра списка ключей и индексов таблицы (рис. 5.55).

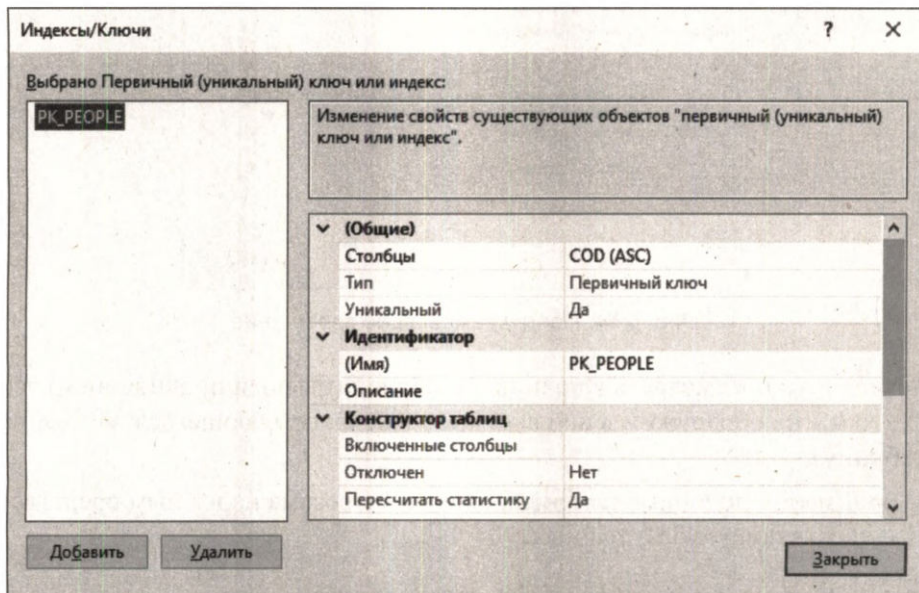


Рис. 5.55. Список ключей и индексов таблицы PEOPLE

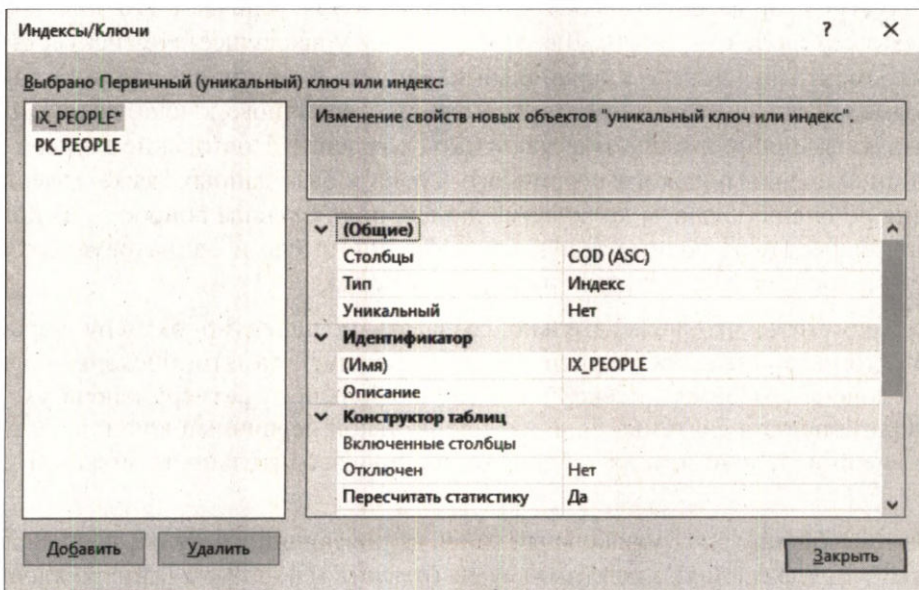



Рис. 5.56. Список характеристик вновь создаваемого ограничения

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 **Добавить**. В этом списке появится новое ограничение, имя которого отмечается символом "звездочка" справа. В нашем примере это имя **IX_PEOPLE***. В правой части окна перечислены его характеристики по умолчанию, которые мы сейчас будем изменять (рис. 5.56).

В правой части окна щелкните мышью по строке **Столбцы**. Справа в строке появится кнопка с многоточием . Щелкните мышью по этой кнопке. Следующим откроется 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 выпадающего списка **Имя столбца** поочередно имена столбцов (заменяя вначале столбец **COD**): **NAME3**, **NAME1**, **NAME2** и **BIRTHDAY**. Значения столбца **Порядок сортировки** для всех элементов ключа можно оставить по возрастанию значений. Список примет следующий вид (рис. 5.57).

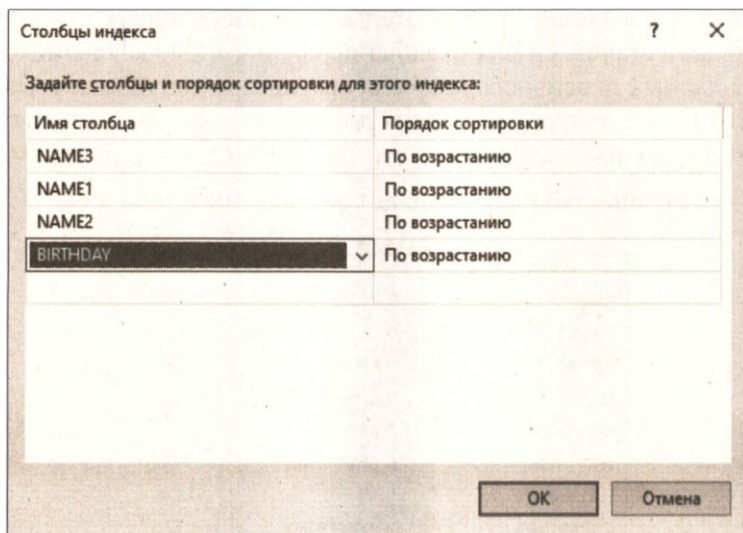


Рис. 5.57. Список элементов столбцов создаваемого ограничения уникального ключа

Щелкните по кнопке **ОК**. В ограничение ключа будут помещены все перечисленные столбцы. Затем в первоначальном списке характеристик создаваемого индекса (пока система рассматривает всю нашу деятельность как создание индекса) щелкните мышью по строке **Тип** и справа из выпадающего списка выберите значение **Уникальный ключ**. Теперь это ограничение уникального ключа, а не индекс.

Задайте новое имя ограничению, щелкнув по строке (**Имя**) и изменив в правой части имя на **UK_PEOPLE**.

Завершите добавление в таблицу уникального ключа, щелкнув мышью в окне по кнопке **Заккрыть**.

Поместив в таблицу выполненное добавление, сохраните ее. Чтобы увидеть сделанные изменения, обновите в **Обозревателе объектов** список, щелкнув по имени таблицы правой кнопкой мыши и выбрав в контекстном меню **Обновить**. После этого последовательно раскройте базу данных **BestDatabase**, папку **Таблицы**, таблицу **PEOPLE** и ее папку **Ключи**. В списке ключей таблицы можно будет увидеть 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 **Обозревателе объектов** по имени таблицы PEOPLE и в контекстном меню выберите элемент **Проект**.

Вначале добавим в таблицу новый столбец CODCTR, который станет внешним ключом, ссылающимся на таблицу стран. Для этого в последней пустой строке списка столбцов таблицы введем имя столбца, CODCTR, в поле **Тип данных** выберем из выпадающего списка значение пользовательского типа данных — char(3). Это 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. Впрочем, это значение и так задано по умолчанию.

Теперь список столбцов таблицы выглядит так, как показано на *рис. 5.58*.

Имя столбца	Тип данных	Разрешить ...
COD	D_INTEGER:int	<input type="checkbox"/>
NAME1	D_CHAR15:varchar(...)	<input checked="" type="checkbox"/>
NAME2	D_CHAR15:varchar(...)	<input checked="" type="checkbox"/>
NAME3	D_CHAR20:varchar(...)	<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	char(3) ▼	<input checked="" type="checkbox"/>

Рис. 5.58. Новый список столбцов таблицы людей

Щелкнем правой кнопкой мыши по любому столбцу таблицы и в контекстном меню выберем элемент **Отношения**. Можно также в главном меню программы выбрать элементы **Конструктор таблиц | Отношения**.

Появится окно, в котором содержится список внешних ключей, связанных с данной таблицей (*рис. 5.59*). Здесь представлены не только внешние ключи самой таблицы PEOPLE, но также внешние ключи других таблиц БД, которые ссылаются на первичный ключ этой таблицы.

Для создания нового внешнего ключа нужно в левой нижней части окна щелкнуть мышью по кнопке **Добавить**. В левой части окна в списке появится новое ограничение с именем FK_PEOPLE_PEOPLE*. В этом окне ничего изменять не будем. Щелкните мышью по кнопке **Закрыть**.

Далее следует сохранить выполненные изменения для таблицы и закрыть окно **Проект**. Теперь в окне **Обозреватель объектов** нужно обновить список, чтобы

можно было увидеть внесенные в таблицу изменения. Щелкните правой кнопкой мыши по имени таблицы PEOPLE и в контекстном меню выберите элемент **Обновить**.

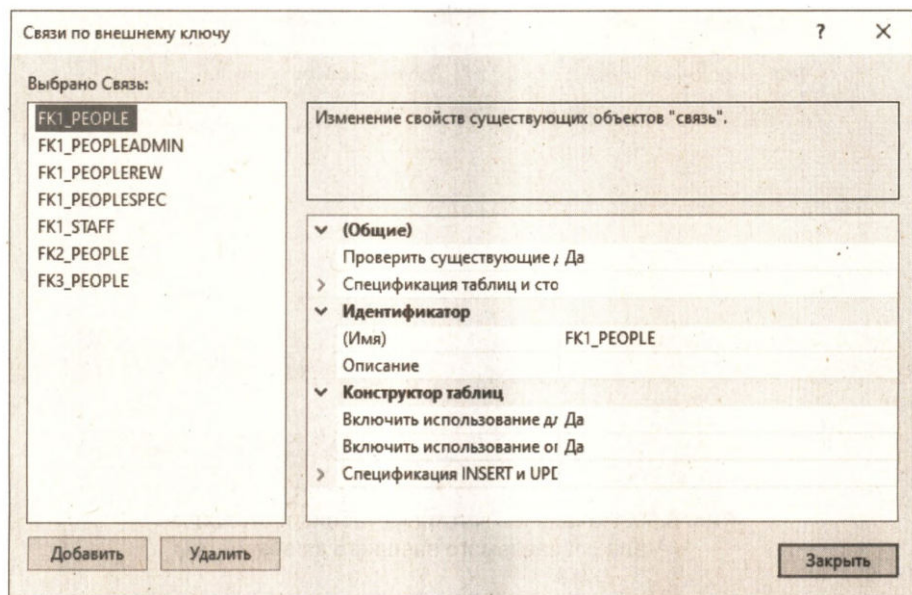



Рис. 5.59. Список внешних ключей, связанных с таблицей PEOPLE

После этого раскройте опять таблицу PEOPLE, раскройте папку ключей, щелкните в появившемся списке ключей правой кнопкой мыши по имени вновь созданного ограничения FK_PEOPLE_PEOPLE и в контекстном меню выберите элемент **Изменить**.

Будет выведено окно, содержащее список всех ключей, имеющих отношение к данной таблице. 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 **(Имя)** и в правой части строки введите новое имя: FK4_PEOPLE.

В этой же части окна выделите мышью строку **Спецификация таблиц и столбцов**. В правой части поля появится кнопка с многоточием . Щелкните мышью по этой кнопке. Появится окно, описывающее таблицы и столбцы, принимающие участие в связях внешних и первичных ключей (рис. 5.60).

В левой части окна (**Таблица первичного ключа**) выберем характеристики родительской таблицы, на первичный ключ которой будет ссылаться наш внешний ключ. Из первого выпадающего списка выберем имя таблицы REFCTR, ниже из выпадающего списка выберем имя столбца этой таблицы, CODCTR. В правой части окна (**Таблица внешнего ключа**) выберем имя столбца, входящего в состав внешнего ключа, — CODCTR.

The dialog box is titled "Таблицы и столбцы". It contains the following fields:

- Имя связи:** A text box containing "FK4_PEOPLE".
- Таблица первичного ключа:** A dropdown menu showing "PEOPLE".
- Таблица внешнего ключа:** A text box containing "PEOPLE".
- Columns:** Two empty text boxes, one under each table name, both containing "COD".
- Buttons:** "OK" and "Отмена" (Cancel) at the bottom right.

Рис. 5.60. Начальная заготовка таблиц и столбцов для создаваемого внешнего ключа

После внесенных изменений окно будет выглядеть, как показано на *рис. 5.61*.

The dialog box is titled "Таблицы и столбцы". It contains the following fields:

- Имя связи:** A text box containing "FK4_PEOPLE".
- Таблица первичного ключа:** A dropdown menu showing "REFCTR".
- Таблица внешнего ключа:** A text box containing "PEOPLE".
- Columns:** Two text boxes. The left one contains "CODCTR". The right one is a dropdown menu showing "CODCTR".
- Buttons:** "OK" and "Отмена" (Cancel) at the bottom right.

Рис. 5.61. Установленные характеристики внешнего ключа

Щелчком мышью по кнопке **ОК**. Теперь остается 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 **Спецификация INSERT и UPDATE** (неудачное название узла; конечно же, должно быть не INSERT, а DELETE). Для строки **Правило удаления** из выпадающего списка выберите элемент **Присвоить NULL**, для строки **Правило обновления** — **Каскадно**. Эти установки соответствуют заданию в языке Transact-SQL предложений

ON DELETE SET NULL

ON UPDATE CASCADE

Напомню, эти предложения означают, что при удалении соответствующей строки родительской таблицы значение внешнего ключа дочерней таблицы устанавливается в NULL, а при изменении значения первичного ключа родительской таблицы это изменение вносится и в значения внешнего ключа всех соответствующих строк дочерней таблицы.

Полученный вид окна показан на *рис. 5.62*.

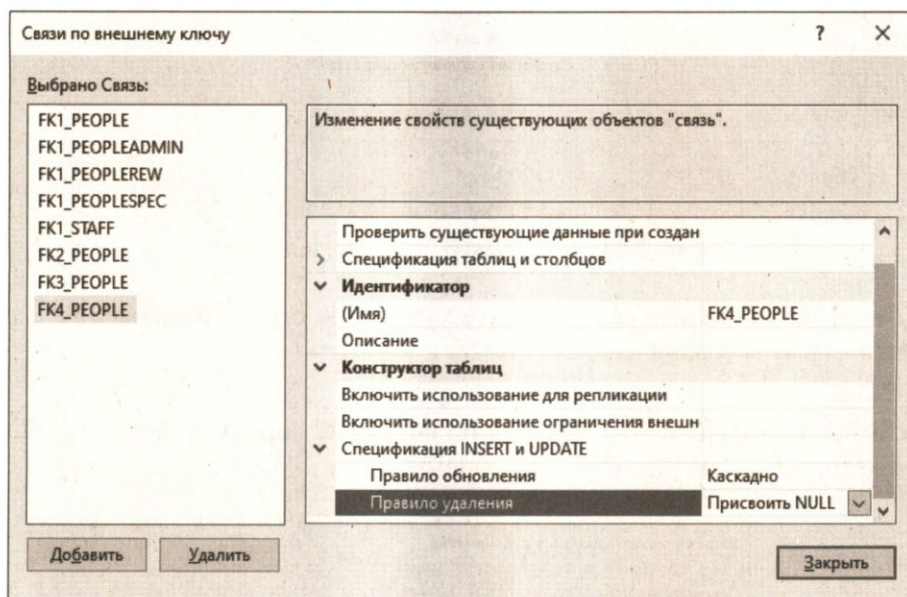


Рис. 5.62. Характеристики созданного нового внешнего ключа для таблицы PEOPLE

Щелкните в этом окне по кнопке **Заккрыть**. 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 признак пола человека, а не то, о чем как-то подумали мои студенты) таблицы PEOPLE.

В окне **Обозреватель объектов** раскройте таблицу PEOPLE, затем папку ограничений, щелкните правой кнопкой мыши по имени ограничения CH_PEOPLE и в контек-

ственном меню выберите элемент **Изменить**. Появится окно **Проверочные ограничения**, содержащее сведения о существующем единственном ограничении для столбца SEX этой таблицы (рис. 5.63).

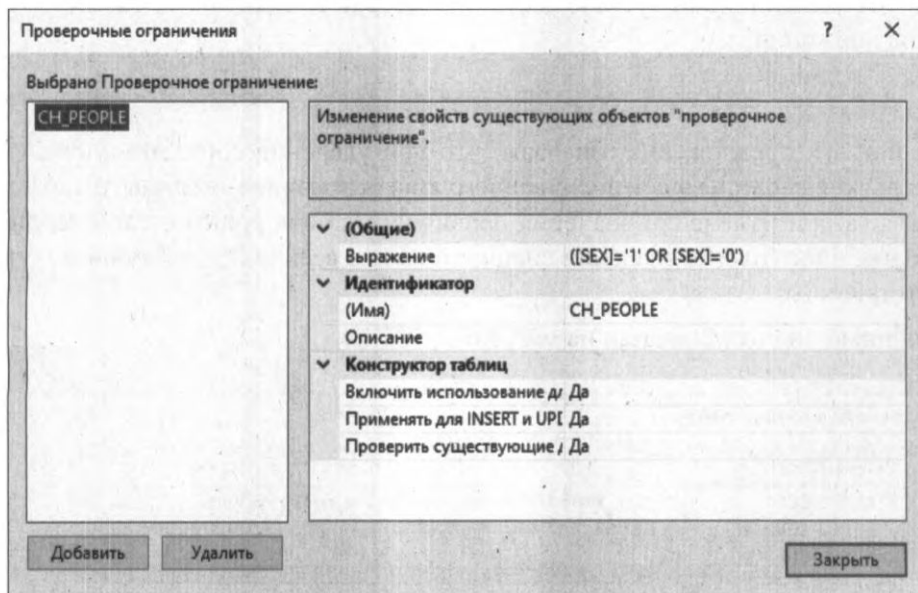



Рис. 5.63. Ограничение CHECK таблицы PEOPLE

Чтобы изменить ограничение, щелкните мышью по строке **Выражение**. 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. 5.64).

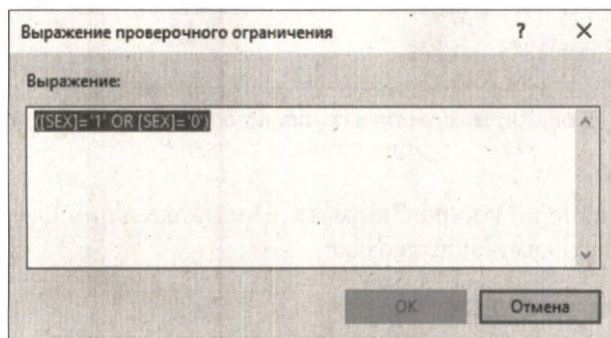


Рис. 5.64. Формула ограничения CHECK таблицы PEOPLE


Измените формулу, 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 **ОК**. В предыдущем окне щелкните по кнопке **Заккрыть**. Сохраните обычным образом изменения таблицы.

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 **Изменить**. Появится окно **Проверочные ограничения**, как было показано на *рис. 5.63*. Чтобы добавить новое ограничение щелкните по кнопке **Добавить**. В списке ограничений появится ограничение с именем **СК_PEOPLE***. В правой части окна замените это имя на **CH2_PEOPLE**. Щелкните мышью по строке **Выражение**, щелкните по кнопке  справа от формулы. В появившемся окне **Выражение проверочного ограничения** введите выражение для проверки значения столбца **CODCTR**:

```
(SELECT COUNT(*) FROM REFCTR R WHERE R.CODCTR = CODCTR) > 0
```

Это условие (на самом деле излишнее) требует, чтобы в таблице **REFCTR** присутствовала строка, первичный ключ которой (**CODCTR**) равен вводимому или изменяемому значению столбца **CODCTR** таблицы **PEOPLE**. Излишним 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, применив функцию **exists()**:

```
exists(SELECT * FROM REFCTR R WHERE R.CODCTR = CODCTR)
```

Щелкните в окне **Выражение проверочного ограничения** по кнопке **ОК**. После этого закройте окно **Проверочные ограничения**, щелкнув по кнопке **Заккрыть**. Попробуйте сохранить изменения таблицы. Вы получите сообщение, что в ограничении **CHECK** недопустимо наличие 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 **Проект** для таблицы **REFCTR**, щелкните правой кнопкой мыши по имени любого столбца и в контекстном меню выберите элемент **Проверочные ограничения**. Появится окно ограничений **CHECK**, в котором не будет ни одного ограничения. Щелкните мышью по кнопке **Добавить**. В списке появится ограничение с именем **СК_REFCTR**. Измените в поле (**Имя**) имя ограничения на **CH1_REFCTR**. В поле **Выражение** введите выражение для проверки числа символов столбца **CODCTR**:

```
(LEN(CODCTR) > 1)
```

В поле **Описание** введите дополнительное текстовое описание, например:

Проверка на количество символов кода страны

В результате окно списка ограничений **CHECK** примет следующий вид (*рис. 5.65*).

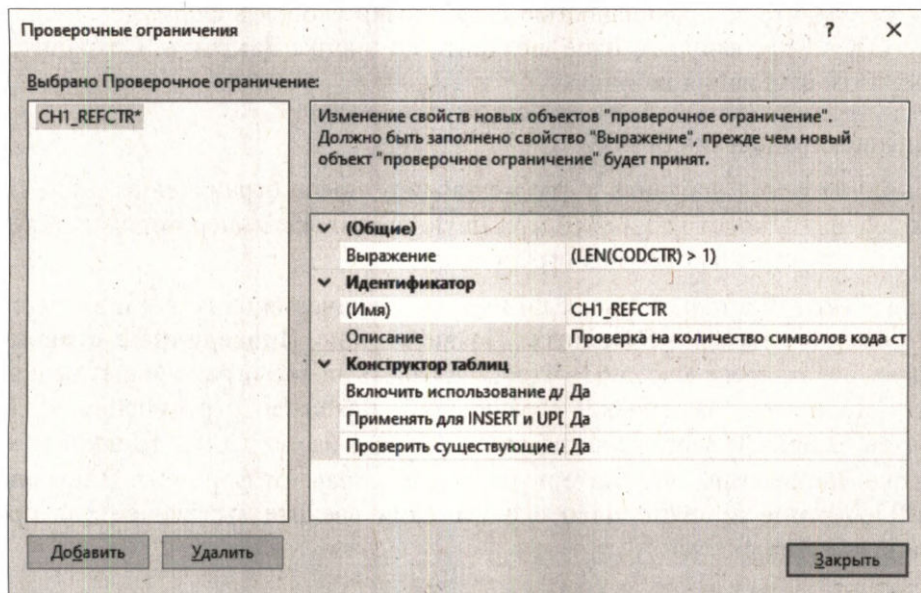


Рис. 5.65. Созданное ограничение CHECK таблицы REFCTR

В этом окне щелкните по кнопке **Закреть**. Сохраните изменения таблицы REFCTR.

Можете проверить работоспособность нового ограничения. В окне **Обозреватель объектов** щелкните правой кнопкой мыши по имени таблицы и в контекстном меню выберите элемент **Изменить первые 200 строк**. Попробуйте изменить код любой страны, оставив один символ. Вы получите сообщение об ошибке.

5.8.2.7. 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. 5.54.

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 присутствует 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),
```

Во вкладке **Проект** удалим столбец NAME1, щелкнув по его имени правой кнопкой мыши и выбрав в контекстном меню элемент **Удалить столбец**. При попытке сохранения таблицы появится диалоговое окно (рис. 5.66).

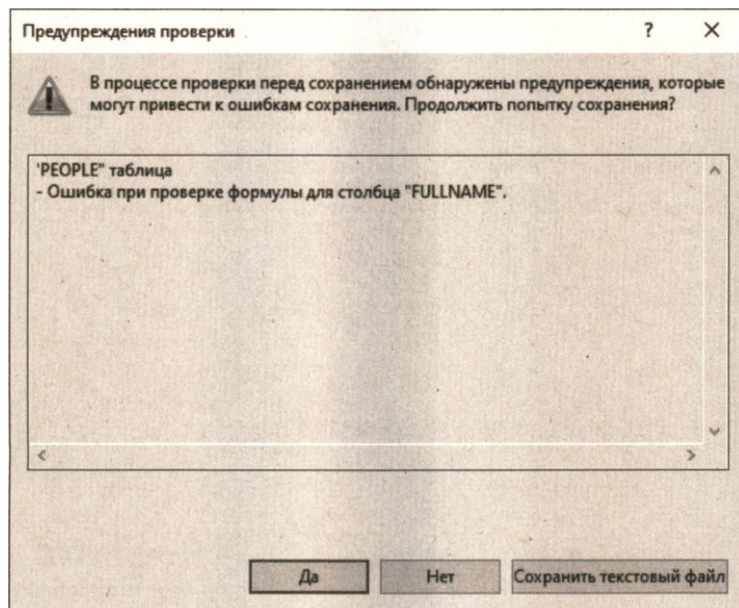


Рис. 5.66. Диалоговое окно, сообщающее о наличии вычисляемого столбца

Здесь выдается сообщение о возникновении ошибки при проверке достоверности формулы для вычисляемого столбца FULLNAME.

При щелчке мышью по кнопке **Да** (что означает требование продолжить попытку сохранения изменений) появляется следующее окно, сообщающее о невозможности изменения таблицы (рис. 5.67).

Здесь остается только щелкнуть мышью по кнопке **ОК**. Столбец не будет удален.

Понятно, чтобы удалить такой столбец, нужно вначале исключить его из формул получения вычисляемых столбцов. Давайте сделаем такую процедуру.

Щелкните мышью по столбцу FULLNAME. В нижней части окна во вкладке **Свойства столбца** раскройте группу **Спецификация вычисляемого столбца**. В строке **(Формула)** содержится 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> или перейдите к другому элементу этой вкладки.

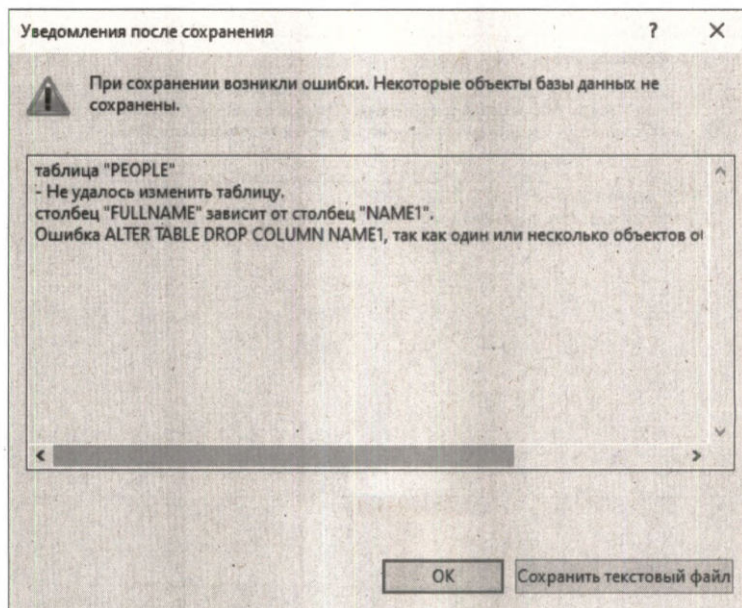


Рис. 5.67. Диалоговое окно, сообщающее о невозможности изменения таблицы

Сохраните изменения в таблице. Вначале вы получите диалоговое окно (рис. 5.68).

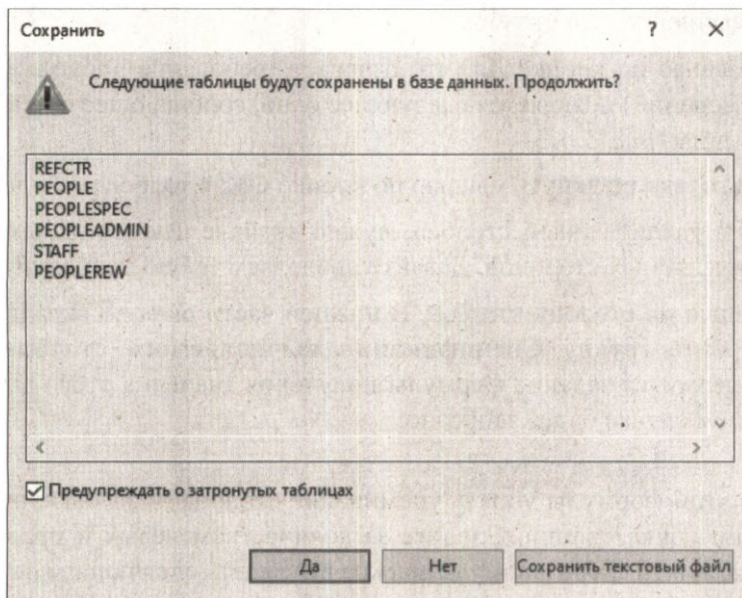


Рис. 5.68. Диалоговое окно, содержащее список изменяемых таблиц

Здесь перечисляются имена таблиц, в которые будут внесены изменения (я опять не понимаю, почему помимо нашей таблицы должны быть изменены другие таблицы). Если вы щелкните по кнопке **Сохранить текстовый файл** и сохраните файл на диске, то ничего интересного там не увидите. Будут перечислены имена таблиц.

Щелкните по кнопке **Да**. Изменения таблицы будут сохранены в базе данных.

Теперь скажем 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.

5.8.2.8. 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 **Проект** для таблицы PEOPLE. Щелкните правой кнопкой мыши по любому столбцу таблицы и в контекстном меню выберите элемент **Проверочные ограничения**. Появится уже хорошо нам знакомое окно, описывающее все ограничения CHECK этой таблицы (рис. 5.69).

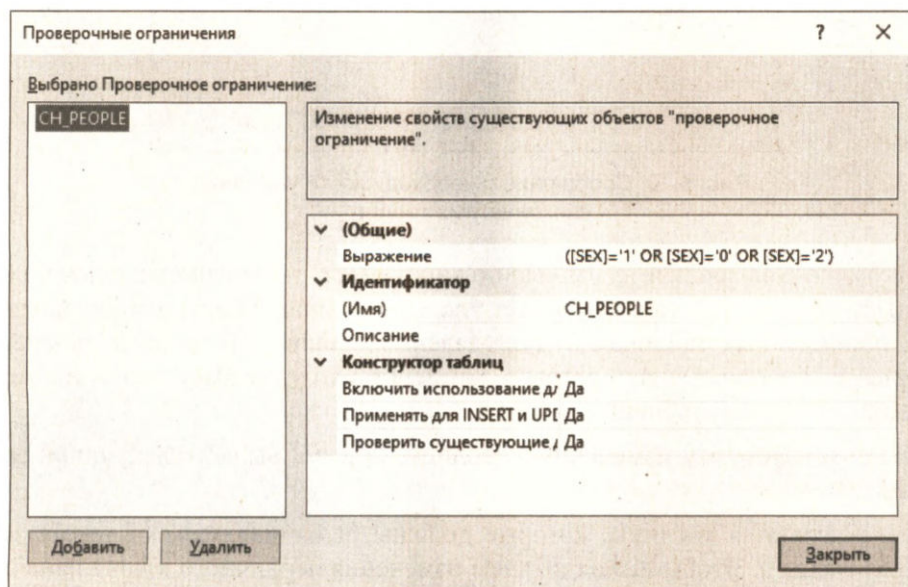


Рис. 5.69. Список ограничений таблицы PEOPLE

Чтобы удалить ограничение `CH_PEOPLE`, нужно, выделив это ограничение, щелкнуть мышью по кнопке **Удалить** в левой нижней части окна. Ограничение пропадет из списка. Затем щелкните по кнопке **Закрыть** 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 проблем. В окне **Проект** 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 **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 потребуется удалить.

Например, попробуйте удалить первичный ключ таблицы `REFREG`. Система выдаст сообщение, как показано на *рис. 5.70*. Здесь спрашивается, хотите ли вы удалить соответствующие отношения.

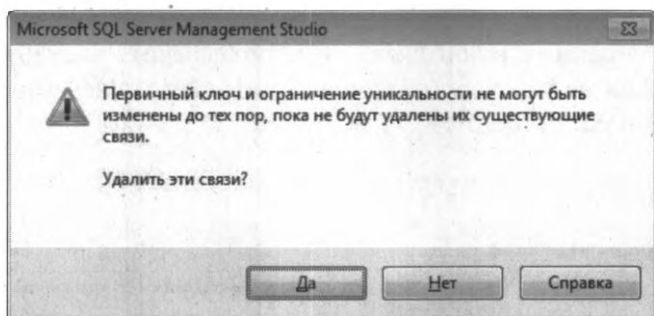


Рис. 5.70. Сообщение о необходимости удаления связей других таблиц

Если в этом окне вы щелкните мышью по кнопке **Нет**, то первичный ключ не будет удален. 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 эти действия будут выполнены только при сохранении изменений таблицы.

Когда вы сохраняете эту измененную таблицу, система выдаст следующий запрос (*рис. 5.71*).

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 **Да**.

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.

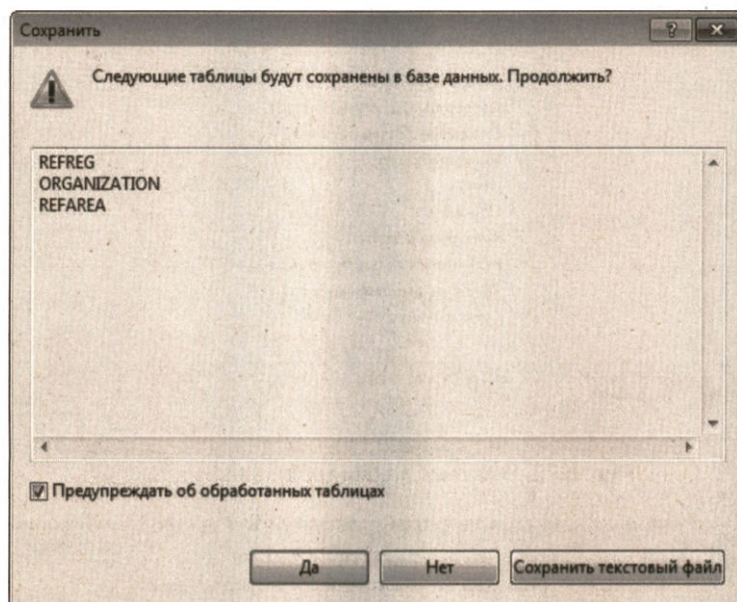


Рис. 5.71. Сообщение об изменении других таблиц

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 ORGACTIV (виды деятельности организаций), ссылающийся на таблицу REFACTIV (справочник видов деятельности).

В окне **Проект** для таблицы ORGACTIV щелкните правой кнопкой мыши по любому столбцу и в контекстном меню выберите элемент **Отношения**. Появится окно, в котором перечисляются все внешние ключи данной таблицы (рис. 5.72).

Для удаления внешнего ключа в левой части таблицы выберите нужный ключ и щелкните по кнопке **Удалить**. Закройте окно, щелкнув мышью по кнопке **Заккрыть**. Сохраните таблицу ORGACTIV. При сохранении таблицы появится предупреждающее окно (рис. 5.73). Несмотря на это сообщение, никаких изменений в таблице REFACTIV на самом деле выполнять не требуется.

При щелчке по кнопке **Да** изменения таблицы будут сохранены в базе данных.

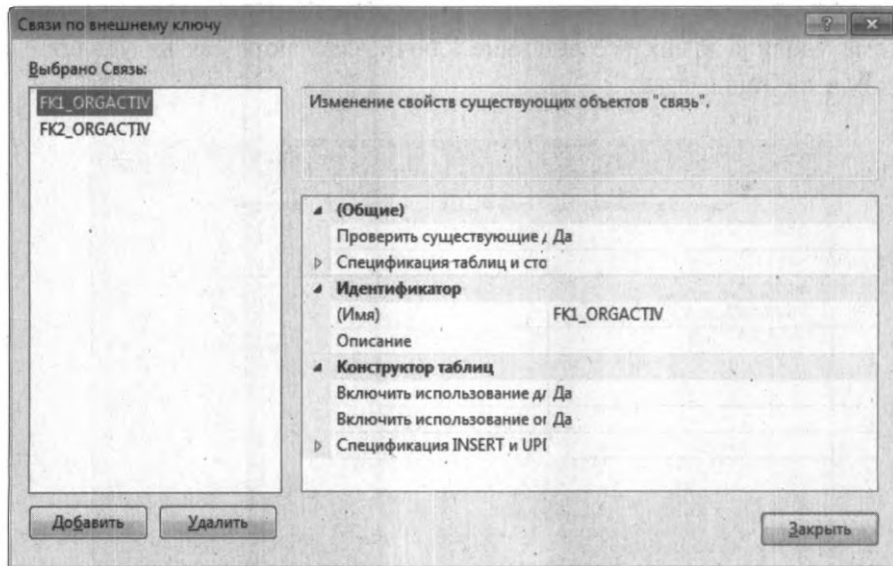


Рис. 5.72. Внешние ключи таблицы ORGACTIV

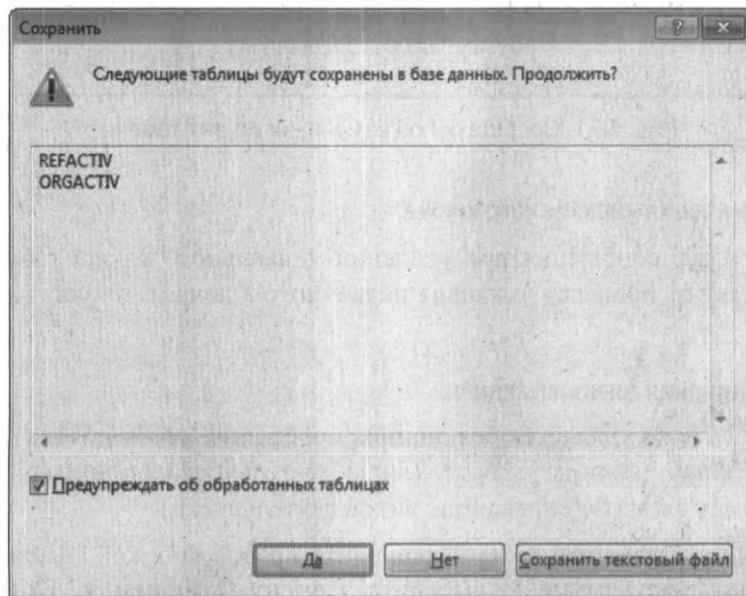


Рис. 5.73. Предупреждающее окно

5.8.3. Построение диаграммы базы данных

В Management Studio есть средство построения диаграммы созданной базы данных, в которой в виде прямоугольников отображаются выбранные из списка таблицы и стрелками указываются ссылки от внешнего ключа дочерней таблицы к первичному (уникальному) ключу родительской таблицы.

Создадим диаграмму для базы данных BestDatabase. В **Обозревателе объектов** раскройте БД **BestDatabase**, щелкните правой кнопкой мыши по строке **Диаграммы базы данных** и выберите в контекстном меню **Создать диаграмму базы данных**.

Появится окно, где вы можете выбрать те таблицы, которые собираетесь включить в диаграмму (рис. 5.74).

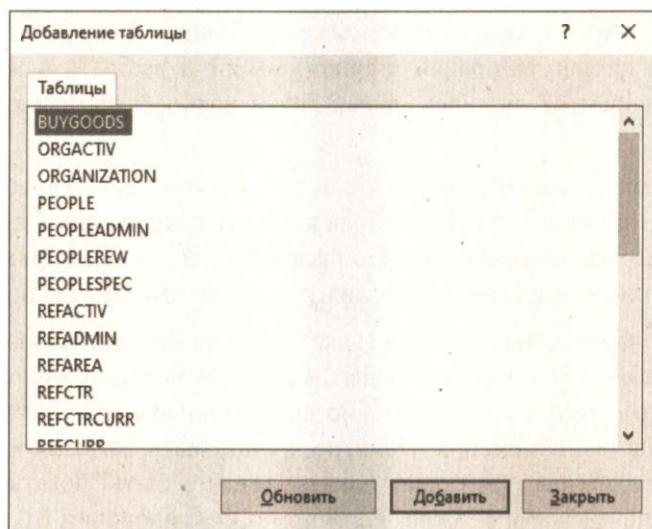


Рис. 5.74. Добавление таблицы в диаграмму

Держа нажатой клавишу <Ctrl>, мышью отметьте таблицы PEOPLE, PEOPLEADMIN, PEOPLESPEC, REFADMIN, REFSPEC и щелкните по кнопке **Добавить**. Будет создана диаграмма (рис. 5.75).

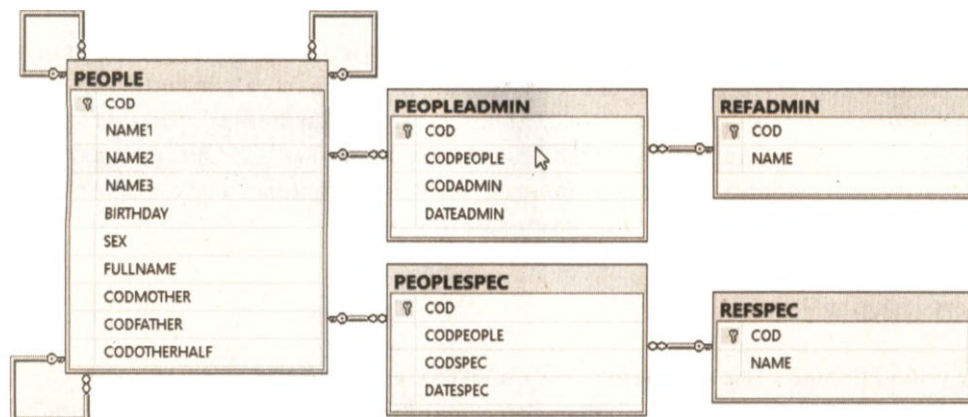


Рис. 5.75. Созданная диаграмма

Для наглядности вы можете мышью переместить любой прямоугольник и любую связь так, чтобы картина производила хорошее впечатление. Размеры прямоугольников тоже можно менять.

Если какая-то таблица вам не нужна на диаграмме, вы можете ее удалить, выделив соответствующий прямоугольник мышью и нажав клавишу <Delete>.

В диаграмме можно вносить некоторые изменения в характеристики столбцов отдельных таблиц, однако этого делать я вам не посоветую.

В созданную диаграмму можно добавлять еще таблицы. Для этого нужно в любом месте диаграммы щелкнуть правой кнопкой мыши и выбрать в меню **Добавить таблицу**. Опять появится окно (см. рис. 5.74), в котором вы выбираете дополнительные таблицы.

Созданную диаграмму можно сохранить в системном каталоге, щелкнув правой кнопкой мыши по заголовку окна и выбрав команду сохранения. После обновления списка в **Обозревателе объектов** можно раскрыть элемент **Диаграммы базы данных**. Откроется список всех созданных диаграмм для этой базы данных.

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 БД 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. Однако, в процессе 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 БД. Они не требуют изменения существующих структур данных, если базу данных вы с самого начала проектировали правильно, т. е. так, как я вам показывал.

Что дальше?

В следующей главе мы поговорим об индексах в SQL Server. Во многих случаях индексы позволяют повысить производительность системы обработки данных. Но при неразумном использовании они также способны резко ухудшить временные характеристики работы с базой данных.

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 2022 существуют следующие виды индексов:

- ◆ обычные индексы (их еще называют реляционными);
- ◆ индексы columnstore;
- ◆ индексы XML;
- ◆ пространственные индексы.

Как уже упоминалось, во многих случаях наличие индексов для таблиц БД 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, соответствующие 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 реализует действительно оптимальный алгоритм поиска данных, основываясь на наличии индексов, количестве записей и на статистических данных.

При отсутствии индексов выборка данных из таблицы выполняется сканированием. В этом случае просматриваются физические страницы БД, 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`.

Перечислим некоторые наиболее интересные столбцы, получаемые из системного представления `sys.indexes`, которое отображает все индексы базы данных:

- ◆ **object_id** — идентификатор объекта БД, которому принадлежит индекс. Для того чтобы найти имя объекта БД по его идентификатору, используется функция `OBJECT_NAME()`, которой передается в качестве параметра идентификатор. Найти идентификатор объекта по его имени позволяет функция `OBJECT_ID()`. Просмотреть объекты БД можно посредством системного представления `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. Вот некоторые его столбцы:

- ◆ **object_id** — идентификатор объекта БД, которому принадлежит индекс описываемого столбца.
- ◆ **index_id** — идентификатор индекса.
- ◆ **index_column_id** — идентификатор столбца индекса.

Представление `sys.xml_indexes` служит 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`.

- ◆ **xml_index_type** — тип индекса: 0 — первичный XML-индекс, 1 — вторичный XML-индекс, 2 — выборочный XML-индекс, 3 — вспомогательный выборочный XML-индекс.
- ◆ **xml_index_type_description** — текстовое описание типа индекса.

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 приведен в листинге 6.1.

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>] ...) ]
}
```

ЗАМЕЧАНИЕ ПО СИНТАКСИСУ

Обратите внимание, что в одном операторе присутствуют два совершенно разных по смыслу предложения `ON`. Одно определяет имя объекта, для которого создается индекс, другое предназначено для задания характеристик секционированности индекса. Вообще говоря, спутать их невозможно, однако совпадение имен многим не нравится. Привердничают они.

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.

6.2.1.1. 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 БД и имя схемы, отделяя их символом точки. Если в операторе не указано имя базы данных, то индекс создается для объекта, присутствующего в текущей БД (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 БД 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.

6.2.1.2. Кластерный и некластерный индексы

Если указано ключевое слово `CLUSTERED`, то создается кластерный индекс. Если задано `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 БД 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 много строк, то процесс создания такого индекса может потребовать немалого времени.

Рекомендуется кластерный индекс создавать прежде, чем будут создаваться другие индексы. В противном случае это приведет к существенным затратам на пересоздание всех остальных индексов.

6.2.1.3. 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 представления) типа данных `XML` можно создавать отдельные индексы `XML`, о чем 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 приблизительным 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 размеща-

ются не все данные одной строки таблицы, а лишь отдельные ее столбцы, указанные в этом предложении.

Предложение **WHERE**

В операторе создания индекса можно указать предложение **WHERE**, в результате получится "фильтрованный" индекс, в состав которого включаются не все строки таблицы, а только некоторые, отвечающие условиям, заданным в этом предложении. Такой фильтрованный индекс не может быть кластерным.

Условие в предложении **WHERE** может быть довольно сложным. В нем допустимы операции сравнения, оператор **IN**, определяющий вхождение значения в указанный список, логические операции конъюнкции, дизъюнкции и отрицания. В условии нельзя использовать в качестве любого литерала неизвестное значение **NULL**. Для проверки на такое значение следует применить конструкции **IS NULL** (является неизвестным значением) или **IS NOT NULL** (не является таким значением).

Задать условия фильтрации для нашей таблицы **PEOPLE** можно, например, в таком виде:

```
WHERE BIRTHDAY > '01.01.1900'
```

Здесь будут индексироваться только строки с данными о людях, 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**) можно указать, где должны располагаться данные индекса:

```
[ ON { <схема секционирования> (<разделяющий ключ>
    | <файловая группа>
    | "default"
  }
]
```

Существуют три варианта размещения индекса в базе данных (как и при создании таблиц).

1. При указании параметра "default" система разместит данные индекса в файловой группе по умолчанию.
2. Если задано имя файловой группы, то индекс будет размещаться в файлах этой файловой группы.
3. Можно создать секционированный индекс, задав имя схемы секционирования и указав в скобках разделяющий ключ. Создание секционированного индекса очень похоже на создание 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 предназначено для создания 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**.

6.2.1.4. Задание параметров индекса

В предложении **WITH** можно перечислить параметры (опции) создаваемого индекса. Эти параметры также применяются 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 разреженного индекса обязательно должен присутствовать параметр **FILLFACTOR**, задающий процент разрежения:

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.

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, когда может происходить добавление данных, влияющее на индекс, имеет смысл уменьшить процент заполнения страницы.

Следует уточнить, что значение этого параметра влияет только на процесс *первоначального* создания или пересоздания индекса. При работе с таблицей процент заполнения отдельных страниц индекса может измениться. В этом случае для восстановления первоначального значения процента заполнения страницы индекса целесообразно изменить 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 БД **tempdb**. Иногда это может сократить время работы системы, если временная БД **tempdb** находится на физическом носителе, отличном от того, где располагается база данных (файлы БД), с которой выполняется работа. Значение **OFF** (значение по умолчанию) — промежуточные данные хранятся в той же самой БД, 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 при дублировании 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, так как оптимизатор запросов не будет располагать достоверными сведениями, позволяющими ему сформировать наиболее эффективный запрос к БД.

Если задано `OFF` (значение 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 указанного объекта индекс с тем же именем. Задает, следует ли системе удалить и перестроить существующий индекс. `ON` — существующий индекс удаляется и перестраивается, `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.

ЗАМЕЧАНИЕ

Нужно сказать, что в любом случае новый индекс лучше создавать в то время, когда иные процессы не осуществляют никаких других действий над соответствующим объектом.

- ◆ `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 }` — как и в предыдущем параметре, задает возможность блокировки страниц (там задавалась блокировка строк) при обращении к индексу. Значение `ON` (значение по умолчанию) разрешает блокировку страниц. `OFF` — запрещает блокировки.

- ◆ **MAXDOP** = <максимальный уровень параллельности> — позволяет установить максимальное число задействованных 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 число процессоров, выполняющих параллельные операции. В реальности может функционировать меньше процессоров, чем указано в параметре. Значение 0 (значение по умолчанию) задает применение всех существующих процессоров или, в зависимости от текущей нагрузки, меньшее их число.

- ◆ **DATA_COMPRESSION** = { NONE | ROW | PAGE }
[ON PARTITIONS (<номер секции> [, <номер секции>] ...)] — позволяет задать режим сжатия данных для индекса:

- если указано NONE, то сжатие данных отсутствует;
- 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 через запятую номера секций, к которым относится указанный режим сжатия (NONE, ROW, PAGE). Для таких индексов параметр DATA_COMPRESSION может повторяться произвольное число раз. Хотя понятно, что реально может потребоваться не более трех повторений — по одному на каждый режим сжатия.

Для того чтобы результаты наших с вами действий с индексами не очень отличались, рекомендую перед выполнением примеров заново создать БД BestDatabase и заполнить ее данными. Нужно последовательно выполнить 11 скриптов, которые вы скачали с сайта издательства (см. приложение 6).

В *примере 6.1* создается некластерный индекс для столбца, содержащего краткое название 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;
GO
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 БД BestDatabase, выполнив скрипт *примера 6.2*.

Пример 6.2. Отображение списка индексов БД BestDatabase

```
USE BestDatabase;  
GO  
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 200 строк. Вот его маленький фрагмент:

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. В результате выборки данных формируется набор данных. Однако результаты представлений не хранятся в базе данных. Они каждый раз получаются вновь при обращении к представлению. К нему можно обращаться как к обычной таблице.

Результат выполнения представления можно сделать хранимым в БД, создав для представления уникальный кластерный индекс. В этом случае набор данных, получаемый при вызове представления, хранится в БД как и при обычном кластерном индексе. 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 ситуациях позволяет повысить производительность системы.

Представления мы будем более подробно рассматривать в *главе 9*.

6.2.3. Создание columnstore индекса

Индексы columnstore появились в версии SQL Server 2012. Основные отличия такого индекса от обычного, "реляционного", классического (rowstore) заключаются в способе формирования и в форме хранения.

При создании обычного индекса группируются и сохраняются индексные данные для строк. Для индекса columnstore выполняется группировка и сохранение данных для столбцов. При этом осуществляется сжатие данных. Для некоторых типов запросов к БД такая структура индекса может сильно повысить производительность. Подобные запросы распространены 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. Индекс можно создавать только для таблиц, но не для представлений.

Данные таблицы, для которой создан индекс columnstore, нельзя изменять.

Синтаксис оператора CREATE COLUMNSTORE INDEX приведен в листинге 6.2.

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>
}
```

Полагаю, 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 указывать два параметра индекса: DROP_EXISTING — задает удаление существующего индекса без выдачи сообщения об ошибке; MAXDOP — задает максимальное число 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 и в составе обычного индекса. Кроме того, 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 способно 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 Гб 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.

С другой стороны, 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 приведен в *листинге 6.3*.

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>
}
```

6.2.4.1. 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 должно быть уникальным среди имен всех индексов данной таблицы.

6.2.4.2. 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. Число 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*).

Формирование 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.

6.2.4.3. 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* ключевого слова `FOR`:

- ◆ `PATH` — создается для выражений пути;
- ◆ `VALUE` — создается для значений. Значение может находиться на любом уровне иерархии в тегах документа XML;
- ◆ `PROPERTY` — содержит значения атрибутов.

Вторичные индексы могут повысить производительность системы при выполнении различных типов запросов.

6.2.4.4. Задание параметров индекса

Как и для обычного индекса, в предложении `WITH` можно перечислить параметры создаваемого индекса XML:

- ◆ `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 БД `tempdb`. `ON` — данные сохраняются в базе данных `tempdb`. `OFF` (по умолчанию) — промежуточные данные хранятся в той же БД, что и индекс.
- ◆ `IGNORE_DUP_KEY = OFF` — при задании `OFF`, если в добавляемых данных содержится дублирующее значение, то все добавление данных, осуществляемое в контексте текущей транзакции, отменяется. Другое значение у этого параметра отсутствует. Вообще-то для XML-индекса этот параметр не нужен, поскольку такие индексы не уникальны.
- ◆ `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), статистические данные по индексу будут пересоздаваться автоматически.

- ◆ `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` — блокировка запрещена.
- ◆ `ALLOW_PAGE_LOCKS = { ON | OFF }` — задает возможность блокировки страниц при обращении к индексу. `ON` (значение по умолчанию) разрешает блокировку. `OFF` — запрещает.
- ◆ `MAXDOP = <максимальный уровень параллельности>` — устанавливает максимальное число используемых процессоров при выполнении параллельных операций с индексами. Может иметь значение от 0 до 64.

Если задано 1, то допустим только один процессор. Значение, превышающее 1, задает максимальное число процессоров, участвующих 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 DESCRSECONDARY1
  ON REFCTRXML (REGIONDESCR)

```

```

    USING XML INDEX DESCRPRIMARY
    FOR VALUE;
CREATE XML INDEX DESCRSECONDARY2
    ON REFCTXML(REGIONDESCR)
    USING XML INDEX DESCRPRIMARY
    FOR PATH;
CREATE XML INDEX DESCRSECONDARY3
    ON REFCTXML(REGIONDESCR)
    USING XML INDEX DESCRPRIMARY
    FOR PROPERTY;
GO

```

Вначале, как обычно, проверяется существование таблицы REFCTXML, и при ее наличии она удаляется из базы данных. При этом автоматически удаляются и все ее индексы. Если требовалось бы удалить только индекс таблицы, то следовало выполнить оператор:

```

IF EXISTS(SELECT * FROM sys.indexes
          WHERE NAME = 'DESCRPRIMARY')
    DROP INDEX DESCRPRIMARY
    ON REFCTXML;
GO

```

Если в операторе удаляется первичный индекс XML, то автоматически удаляются и все вторичные.

Далее для столбца REGIONDESCR таблицы REFCTXML, 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",
       CAST(xml_index_type AS CHAR(9)) AS "XML Type",
       CAST(xml_index_type_description AS CHAR(14)) AS "XML Type Descr"
FROM sys.xml_indexes;
GO

```

Вы получите следующий список:

Name	Type	Secondary Type	Description	XML Type	XML Type Descr
DESCRPRIMARY	XML	NULL	NULL	0	PRIMARY_XML

DESCRSECONDARY1	XML	V	VALUE	1	SECONDARY_XML
DESCRSECONDARY2	XML	P	PATH	1	SECONDARY_XML
DESCRSECONDARY3	XML	R	PROPERTY	1	SECONDARY_XML

Для отображения характеристик индексов XML существуют и другие системные средства.

Тип данных XML и все средства работы с такими данными довольно сложны. 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. Могу порекомендовать, например, такую книгу: Хантер Д., Рафтер Д., Фаусетт Д. XML. Базовый курс. М.: Диалектика, 2018.

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 можно включать только один столбец.

Синтаксис оператора CREATE SPATIAL INDEX для создания пространственного индекса приведен в листинге 6.4.

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>] ... }
                      )

<ключевое слово> ::= { 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>
    }

```

В процессе индексирования пространственного столбца выполняется декомпозиция пространства (содержимого столбца) в сеточную иерархию и так называемая тесселяция.

При декомпозиции пространство преобразуется в четырехуровневую сеточную иерархию. Декомпозиция выполняется одинаково для геометрического и географического столбцов и не зависит от используемых единиц измерения.

Для геометрического типа данных в предложении BOUNDING_BOX указывают четыре координаты ограничивающего прямоугольника.

После ключевого слова GRIDS в позиционном или ключевом формате задают 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 нужны пространственные типы данных и соответствующие индексы, рекомендую обратиться к фирменной документации, а лучше — найти хорошую книгу по этим вопросам. Рекомендую, например, такую книгу: Воронин А. В. Обработка пространственных и тематических данных в геоинформационной системе. М.: Горячая Линия – Телеком, 2021.

6.2.6. Удаление индекса

Оператор `DROP INDEX` удаляет любой индекс (обычный, XML или пространственный), созданный пользователем. 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` приведен в листинге 6.5.

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"
}
]
```

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:

- ◆ `MAXDOP` — как и при создании индекса, позволяет установить максимальное число используемых процессоров при выполнении параллельных операций с индексом. Параметр может иметь значение от 0 до 64.
- ◆ `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 числе строк таблицы) выполняется быстрее.
- ◆ `MOVE TO` — может указываться только при удалении кластерных индексов. Параметр указывает, куда должны перемещаться строки таблицы при удалении индекса. Напомню, что в случае кластерного индекса строки таблицы располагаются в конечных узлах индексного дерева.

Таблицу можно сделать секционированной, строки 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 предусмотрен оператор `ALTER INDEX`. Его синтаксис приведен в *листинге 6.6*.

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 }
}

```

В операторе изменения индекса можно указать имя изменяемого индекса или задать ключевое слово **ALL**, которое указывает, что изменение должно 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 перестроения 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_COMPACTON = ON`, то выполняется сжатие данных больших двоичных объектов `LOB`.

Остальные параметры соответствуют параметрам оператора создания индекса.

6.3. Работа с индексами в диалоговых средствах 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 **Индексы** соответствующей таблицы и в контекстном меню выбрать элемент **Создать индекс**. Появится подменю, в котором будут перечислены возможные для текущей таблицы виды индексов:

- ◆ Кластеризованный индекс.
- ◆ Некластеризованный индекс.
- ◆ Первичный XML-индекс.
- ◆ Вторичный XML-индекс.
- ◆ Пространственный индекс.
- ◆ Некластеризованный индекс columnstore.
- ◆ Кластеризованный индекс и индекс columnstore.

В зависимости от типов данных столбцов, входящих в состав таблицы, недопустимые виды индексов будут представлены строками серого цвета.

Давайте создадим обычный, реляционный индекс для таблицы `PEOPLE` из нашей БД данных `BestDatabase`. Запустите на выполнение Management Studio, в панели **Обозреватель объектов** раскройте папки **Базы данных**, **BestDatabase**, **Таблицы**. Раскройте папку таблицы `dbo.PEOPLE`. Щелкните правой кнопкой по папке **Индексы**. В появившемся контекстном меню выберите элемент **Создать индекс**, а затем **Некластеризованный индекс**. Появится окно **Новый индекс** (рис. 6.1).

Опять появится окно создания индекса с выбранными столбцами (рис. 6.3), в котором можно корректировать список. Выделив столбец, можно переместить его выше по списку (кнопка **Переместить вверх**), ниже (кнопка **Переместить вниз**), удалить выделенный элемент (кнопка **Удалить**) или добавить новые элементы (кнопка **Добавить**).

Здесь также есть возможность изменить упорядоченность индекса по любому столбцу. Для этого нужно в выделенном столбце щелкнуть мышью по полю с заголовком **Порядок сортировки**. После этого из выпадающего списка можно выбрать упорядоченность значений по возрастанию или по убыванию. По умолчанию устанавливается возрастающий порядок.

Чтобы указать, что индекс уникальный, нужно отметить флажок в поле **Уникальный**.

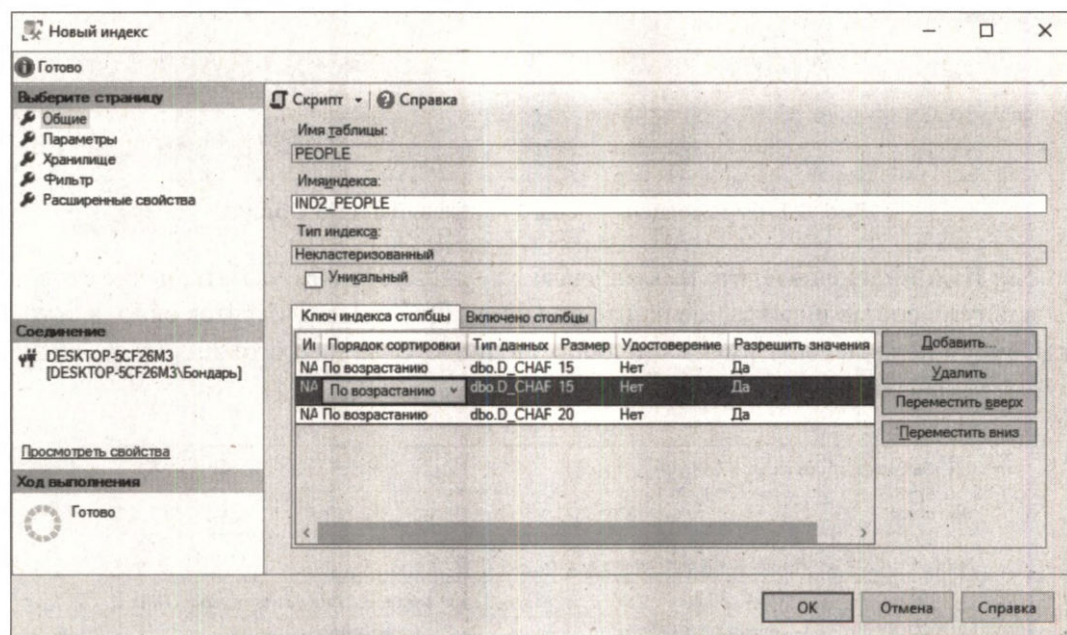


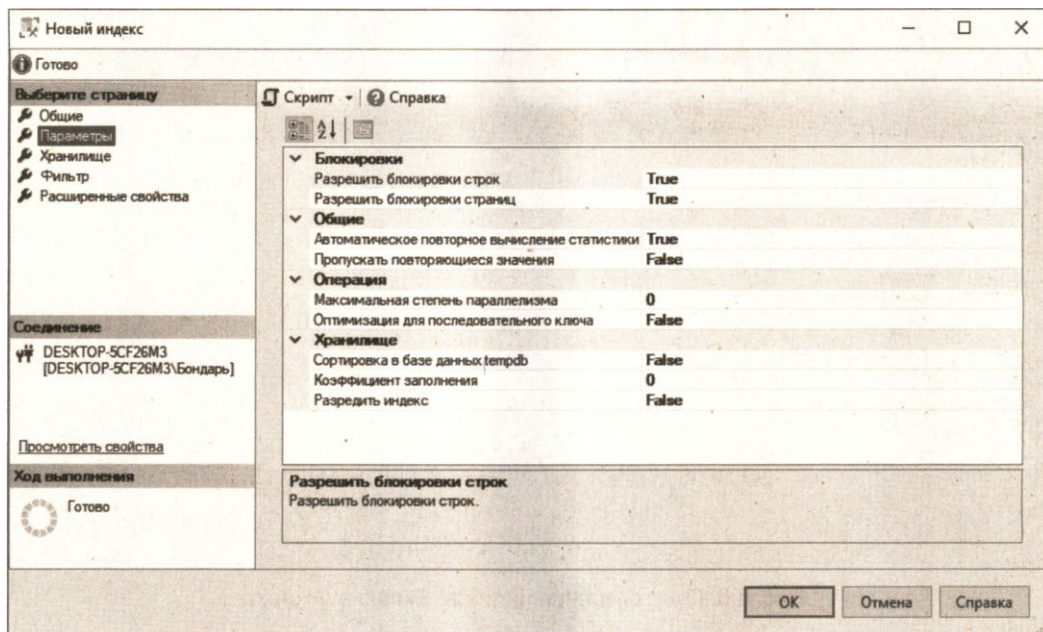
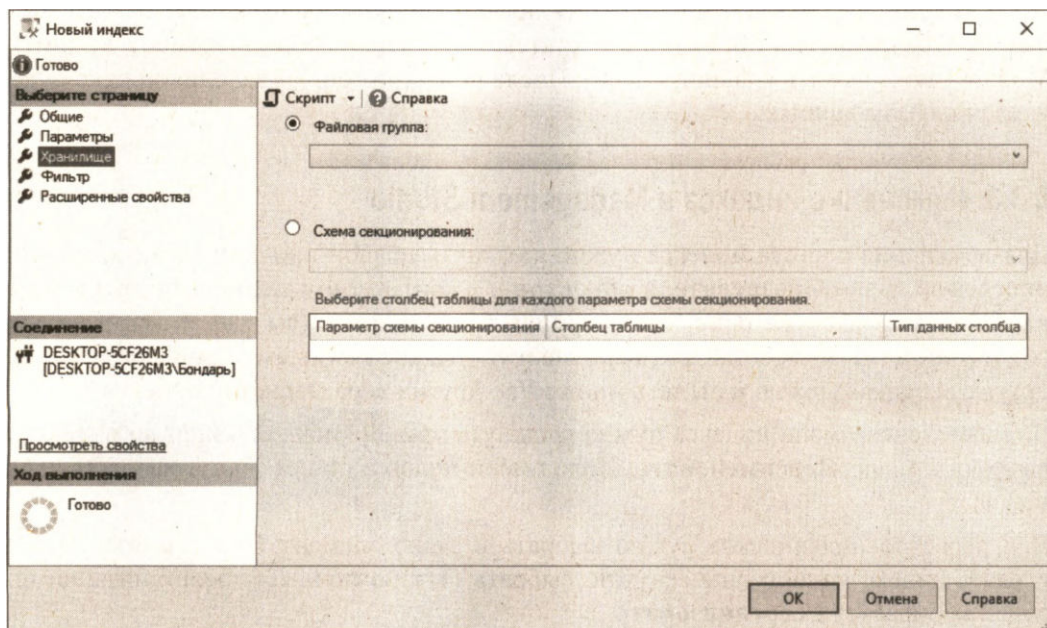
Рис. 6.3. Корректировка списка столбцов индекса

Если перейти во вкладку **Параметры**, то появится список параметров индекса (рис. 6.4).

Все параметры мы уже рассмотрели, когда говорили об операторе создания индекса.

Во вкладке **Хранилище** (рис. 6.5) можно выбрать файловую группу или схему секционирования для секционированного индекса.

Вкладка **Фильтр** (рис. 6.6) позволяет задать условие фильтрации индекса. Здесь указывают выражение, определяющее те строки исходного объекта, которые должны быть проиндексированы.

Рис. 6.4. Окно создания индекса. Вкладка **Параметры**Рис. 6.5. Окно создания индекса. Вкладка **Хранилище**

Для завершения создания индекса нужно щелкнуть мышью по кнопке **ОК**. Новый индекс появится в списке индексов таблицы.

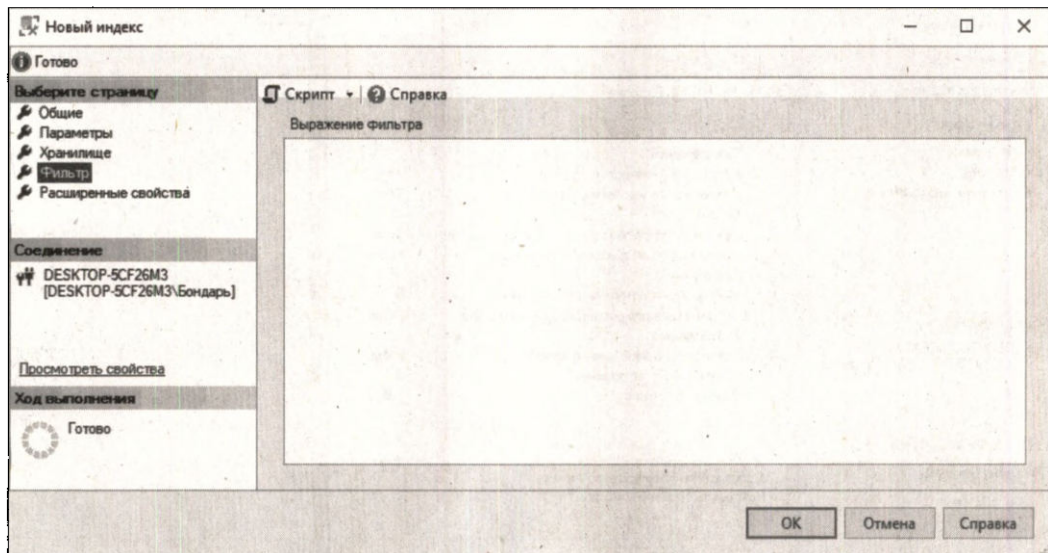


Рис. 6.6. Окно создания индекса. Вкладка **Фильтр**

6.3.2. Удаление индекса в Management Studio

Для удаления индекса нужно щелкнуть правой кнопкой мыши по имени индекса в панели **Обозревателя объектов** и в контекстном меню выбрать элемент **Удалить**. Можно также нажать клавишу . 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 **Обозревателя объектов** и в контекстном меню выбрать **Свойства**. Появится такое же окно, как и при создании индекса (см. рис. 6.3). Здесь есть возможность менять 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 выбрать в меню **Переименовать**. Поле имени индекса станет доступным для изменения.

Для перестроения индекса нужно выбрать в меню элемент **Перестроить**. Чтобы сделать индекс недоступным, нужно выбрать **Отключить**. Для реорганизации индекса выбирается **Реорганизовать**.

Что дальше?

В следующей главе мы рассмотрим операторы, позволяющие добавлять, изменять и удалять данные таблиц БД.

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 БД. Этот набор данных получается при использовании оператора `SELECT`, обращающегося к одной или более реальным таблицам, присутствующим в БД. К этому набору данных можно обращаться по его имени в операторах, выполняющих действия с данными в БД.

Обобщенное табличное выражение может быть рекурсивным, т. е. включать и ссылки на себя.

Синтаксис предложения `WITH` для обобщенного табличного выражения приведен в листинге 7.1.

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.

Число указанных в скобках столбцов должно совпадать с числом столбцов, возвращаемых оператором SELECT в предложении AS. По именам столбцов можно обращаться к соответствующим значениям. Имена столбцов, заданные в обобщенном табличном выражении, должны быть уникальными. Список имен столбцов не обязателен, 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 опять же при помощи оператора SELECT.

Если в предложении WITH присутствуют 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 приведен в листинге 7.2.

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  
} ;
```

В начале оператора может быть указано 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 число либо процент (ключевое слово `PERCENT`) добавляемых строк в этом операторе. Оператор может задавать добавление нескольких строк. При наличии предложения `TOP` число 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 определяться при задании представления, которое должно быть изменяемым. О представлениях мы поговорим в *главе 9*.

Оператор позволяет также добавлять данные в таблицы БД, находящиеся 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 БД, который создается хранимой процедурой `sp_addlinkedserver`. Его также можно создать в *Management Studio*. В результате появляется возможность 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, таблицы Excel, БД Access, Oracle и IBM DB2.

Вместо имени связанного сервера можно задать обращение к функции `OPENDATASOURCE`, которая подключит к источнику OLE DB.

К источнику данных OLE DB также можно обратиться при помощи функций `OPENQUERY` и `OPENROWSET`. В этом случае нужно задать также оператор обращения к данным таблицы связанного сервера, в которые будут добавляться данные.

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, например поведение системы по отношению к триггерам, ограничениям, автоинкрементным первичным ключам, задание характеристик транзакции и ряд других.

В операторе можно в круглых скобках указать список имен 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 имен 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 БД. Это может быть обычная или временная таблица.

Сами значения, помещаемые в столбцы новой строки таблицы, можно задать одним из четырех способов:

- ◆ в предложении `VALUES` с перечислением значений. Это наиболее распространенный вариант;
- ◆ при помощи оператора `SELECT`, который возвращает одну или более строк таблицы. Также часто используемый вариант;
- ◆ при вызове хранимой процедуры оператором `EXECUTE`;
- ◆ присваиванием значений по умолчанию всем столбцам (`DEFAULT VALUES`).

Число значений, возвращаемых операторами или указанных явно, должно соответствовать числу заданных имен столбцов таблицы.

В случае задания предложения `VALUES` можно в одном операторе добавлять множество строк, повторяя через запятую список значений в скобках. В предложении `VALUES` термин `<выражение>` означает и собственно выражение, которое возвращает конкретное значение, и константу.

Рассмотрим несколько примеров помещения данных в таблицы БД `BestDatabase`.

Создадим заново базу данных `BestDatabase` и несколько таблиц, описывающих страны, их регионы, районы. В первом примере даются фрагменты простых операторов, помещающих данные в эти таблицы (*пример 7.1*).

Пример 7.1. Добавление данных в таблицы стран, регионов, районов

```
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

USE BestDatabase;

GO

SET NOCOUNT ON;

BEGIN TRANSACTION;

-- Страны

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 */
  TELCODE VARCHAR(10) /* Телефонный код */
  CONSTRAINT PK_REFCTR PRIMARY KEY (CODCTR)
);

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

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 */
  IND CHAR(6), /* Почтовый индекс региона */
  PHONE CHAR(10), /* Телефонный код региона */
  CONSTRAINT PK_REFREG PRIMARY KEY (CODCTR, CODREG),
  CONSTRAINT FK_REFREG
    FOREIGN KEY (CODCTR) REFERENCES REFCTR (CODCTR)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);

/* 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
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 */
  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
);
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 один оператор `INSERT` осуществлял добавление всех строк районов.

Оператор `SET NOCOUNT ON` предназначен для того, чтобы после каждой операции добавления строки мы не получали сообщение, что одна строка добавлена.

В следующем примере (пример 7.2) добавим несколько записей в таблицу людей.

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. Поэтому явно задавать для него какое-либо значение не нужно.

Пример 7.2. Добавление данных в таблицу людей

```

USE BestDatabase;
GO
SET NOCOUNT ON;

```

```

SET DATEFORMAT dmy;

/*** 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 */
  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
);

/* 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, CODOTHERHALF)
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, CODOTHEHALF)
VALUES ('ПРАСКОВЬЯ', 'СТЕПАНОВНА', 'ПИНТЕРА', '01.01.1946', '1',
(SELECT COD FROM PEOPLE
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 CODOTHEHALF) используется оператор SELECT, выбирающий из базы данных значение кода этой супруги. Однозначно идентифицировать запись позволяют значения фамилии, имени и отчества. В данном случае оператор вернет ровно одно значение, поскольку в БД существует только одна 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 указано предложение WHERE, задающее 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 CTE

```
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 приведен в *листинге 7.3*.

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> ::= += | -= | *= | /= | %= | &= | ^= | |= | =
```

В начале оператора может быть задано обобщенное табличное выражение.

В необязательном предложении TOP указывается число 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, а также таблица на связанном сервере.

Здесь могут использоваться 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 применяются 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 параметр <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 исключающего 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 очень сложное условие или 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 влияют 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.

В следующем *примере 7.5* выполняется обновление данных в таблице персонала STAFF.

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, сторонник гендерного неравенства. 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
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 */
  CODPEOPLE 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
);
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 изменяет строки, относящиеся к мужчинам. Второй — к женщинам. Для выборки множества изменяемых строк используется предложение WHERE, задающее условие отбора необходимых строк таблицы. В этом условии присутствует оператор SELECT, отбирающий записи с нужным значением пола 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 приведен в листинге 7.4.

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>]...) ] ;
```

Никаких вопросов этот синтаксис у нас с вами не вызывает. Все предложения нам хорошо известны по предыдущим операторам.

Первым может быть задано обобщенное табличное выражение. В предложении *TOP* указывается число 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) = '1') AND
         CODORG = 11;
DELETE FROM STAFF
  WHERE ((SELECT SEX FROM PEOPLE
           WHERE PEOPLE.COD = STAFF.CODPEOPLE) = '0') 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 с помощью оператора **TRUNCATE TABLE**. При этом удаление происходит намного быстрее. Синтаксис оператора приведен в *листинге 7.5*.

Листинг 7.5. Синтаксис оператора TRUNCATE TABLE

```
TRUNCATE TABLE
  [[<имя базы данных>.]<имя схемы>.]<имя таблицы> ;
```

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 рассмотрим в несколько упрощенном варианте синтаксис оператора, который приведен в *листинге 7.6*.

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
}
```

Как и в других операторах, в начале оператора может идти описание обобщенного табличного выражения.

После ключевого слова `TOP` можно указать, что изменениям подвергается только указанное число 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, на основании данных которой будет изменяться целевая таблица. 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 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 могут присутствовать два предложения `WHEN MATCHED`. Второе будет выполняться только в том случае, если не было выполнено первое. При наличии 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 вспомним сведения из предыдущих глав.

Задача в том, чтобы создать БД `DoubleDatabase`, которая будет копировать некоторые сведения из основной БД `BestDatabase`. Эта вторая база данных будет путешествовать по всему миру на ноутбуке одного активного коммивояжера. Он будет вносить изменения в БД в соответствии с информацией, полученной в поездках. После его приезда на родину нужно выполнить синхронизацию обеих баз данных.

Вначале создадим дублирующую БД в каталоге DoubleDatabase и в ней три таблицы (пример 7.7).

Пример 7.7. Создание дублирующей БД

```
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 БД (*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 БД:

```

INSERT INTO ORGACTIV (CODACT, CODORG)
  SELECT CODACT, CODORG FROM BestDatabase.dbo.ORGACTIV;

```

Можно отобразить этот список. В результате получим:

```

CODACT CODORG TEXT
-----

```

2010	8	NULL
2020	8	NULL
2021	8	NULL
2022	8	NULL
2023	8	NULL
2029	8	NULL

Точно такие же значения на текущий момент будут и в основной БД.

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 БД выглядят так:

CODACT	CODORG	TEXT
2020	8	NULL
2021	8	NULL
2022	8	NULL
2023	8	NULL
2010	9	NULL
2029	9	NULL
1812	9	NULL

По приезду 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` БД `BestDatabase`. Исходная таблица находится в другой базе данных. Эта

таблица задается после ключевого слова `USING`. Для таблицы указываются имя БД и имя схемы по умолчанию. Здесь же таблице присваивается псевдоним `FIRSTTAB`, чтобы далее не возникла двусмысленность 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 такие строки удаляются, как указано в предложении `WHEN NOT MATCHED BY SOURCE`.

Отобразите строки таблицы `ORGACTIV` в основной БД. Мы видим, что данные в обеих базах данных совпадают.

Что дальше?

В следующей *главе 8* мы, наконец, начнем рассматривать важнейший и наиболее сложный оператор выборки данных — `SELECT`.

Выборка данных

- ◆ Выборка данных. Оператор **SELECT**.
- ◆ Оператор **UNION**.
- ◆ Операторы **EXCEPT** и **INTERSECT**.
- ◆ Примеры использования операторов.

8.1. Оператор **SELECT**

Мы уже многократно применяли оператор **SELECT** для получения различных данных из баз данных. Настала пора подробно рассмотреть его синтаксис и примеры использования.

В операторе **SELECT**, так же, как и во многих других операторах DML, можно использовать обобщенное табличное выражение CTE. Начальный синтаксис оператора приведен в *листинге 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>]... ) ] ;
```

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 результирующий набор 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 число либо процент (ключевое слово `PERCENT`) выбираемых строк. Далее идет список выбора, который содержит список элементов, разделенных запятыми. Синтаксис конструкции <элемент выбора> приведен в листинге 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> ]
}
```

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`.

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 или псевдоним таблицы, представления.

Если задано ключевое слово `$IDENTITY`, то в список выбора помещается столбец таблицы или представления со свойством `IDENTITY`. Ключевое слово `$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. Псевдоним также можно указать в этом же операторе, например в выражении `ORDER BY`, задающем упорядочивание результата.

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` указывают 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` приведен в *листинге 8.3*.

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
}
```

В предложении `FROM` исходная таблица может быть задана таблицей или представлением. Задать этот объект можно с помощью всех вариантов, рассмотренных нами в предыдущей *главе 7*. Объект может находиться в том же или в другом экземпляре сервера, он может быть и вне экземпляра сервера SQL. После необязательного ключевого слова `AS` может быть указан псевдоним таблицы (представления), который можно использовать 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 учтены 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`.

Замечательная возможность оператора `SELECT` — *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.

Существуют следующие виды 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 условию соединения значения столбцов правой таблицы. Очень часто в своих реальных разработках я использую этот вид соединения;
 - 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. Предполагаю, что этот вид соединения был создан для программистов левшей;
 - полное внешнее соединение (`FULL OUTER JOIN`) — возвращает все строки левой и правой таблицы.
- ◆ `CROSS JOIN` — возвращает 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. Получив аргумент, функция выделяет 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 называют и условием поиска, приведен в *листинге 8.4*.

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> )
}

```

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 выражение, возвращающее логическое значение "истина", "ложь" или **NULL** для 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**, **CONTAINS** 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 приведены таблицы истинности для логических операций отрицания, дизъюнкции и конъюнкции для трех значений операндов: "истина" (**True**), "ложь" (**False**) и **NULL**. Поскольку операнд может иметь и значение **NULL**, здесь используется трехзначная логика.

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 соответствует правилам логики высказываний.

Первой конструкцией в синтаксисе предиката является <выражение> <операция сравнения> <выражение>.

Выражением может быть любое допустимое выражение языка SQL. Чаще всего это просто имя столбца таблицы. В правой части предложения может присутствовать оператор SELECT, который возвращает ровно одно значение. Примеры мы вскоре рассмотрим.

Операции сравнения 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 число символов
_	Один произвольный символ
[]	Задание диапазона допустимых значений. Подходит любой символ из указанного диапазона. Символы диапазона указываются через дефис, например [a-z]. Это означает, что допустимы 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 использовать шаблонные символы как обычные символы шаблона, а не как шаблонные символы. Если шаблонный символ в строке необходим в роли обычного символа, 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 учитываются 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 на основании содержания (семантики), а не формы текста (синтаксиса).

Конструкция `IN (NOT IN)` задает выбор тех строк, значения столбцов которых находятся в указанном списке или отсутствуют в списке, при задании ключевого слова `NOT`. Элементами в списке могут быть константы, выражения, возвращающие одно значение, или подзапросы, также возвращающие ровно одно значение. Список может содержать один оператор `SELECT`, который возвращает произвольное число значений, например при выборке значений одного столбца из таблицы или представления.

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 число значений *одного* столбца. Такие подзапросы называют в литературе *скалярными*.

Функции `SOME` и `ANY` — это синонимы. Подзапрос также является скалярным, т. е. возвращает произвольное число 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 некоторых 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 `GROUP BY` приведен в листинге 8.5.

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>]... )
```

Часто предложение `GROUP BY` используется при появлении в списке выбора оператора `SELECT` агрегатных функций `AVG` (среднее значение), `COUNT` (подсчет числа 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 служат для формирования отчетов.

Функция `ROLLUP()` позволяет создавать иерархические итоговые данные по уровням элементов, указанных в параметрах функции. `CUBE()` дает возможность создавать итоги с использованием перекрестных вычислений. `GROUPING SETS()` задает несколько вариантов группировки данных в одном запросе. Примеры мы рассмотрим далее в этой главе.

Предложение `HAVING` позволяет ограничить число выбранных строк с учетом предложения `GROUP BY`. Если же `GROUP BY` не присутствует в операторе, то `HAVING` применяется как предложение `WHERE` для ограничения числа строк, возвращаемых оператором.

Предложение `ORDER BY` задает упорядочение строк результата запроса. Синтаксис предложения приведен в листинге 8.6.

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 ]
```

Здесь <элемент упорядочения> — это имя или псевдоним столбца либо номер столбца в списке выбора, по которому выполняется упорядочение. Если столбец имеет строковый тип данных, то можно указать используемый для упорядочения порядок сортировки после ключевого слова `COLLATE`. 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 число возвращаемых строк. Ключевые слова `FIRST` и `NEXT` являются синонимами и предназначены 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 мы также не рассматриваем в данной книге.

8.2. Оператор *UNION*

Оператор `UNION` позволяет объединить в один выходной набор данных результаты выборки несколькими операторами `SELECT`. Синтаксис оператора приведен в *листинге 8.7*.

Листинг 8.7. Синтаксис оператора `UNION`

```
<оператор UNION> ::= <запрос> UNION [ ALL ] <запрос>
[ UNION [ ALL ] <запрос> ] ... ;
```

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. `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. `INTERSECT` возвращает недублированные строки первого запроса, которые соответствуют строкам, полученным из второго запроса. Это операция пересечения двух множеств.

Синтаксис обеих операций приведен в *листинге 8.8*.

Листинг 8.8. Синтаксис операторов `EXCEPT`, `INTERSECT`

```
<оператор EXCEPT, INTERSECT> ::=
<запрос> { EXCEPT | INTERSECT } <запрос>;
```

8.4. Примеры выборки данных

Теперь рассмотрим основные возможности оператора `SELECT` на примере нашей БД `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.

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 применить функцию `IIF()`. Кроме того, код человека является искусственным первичным ключом и в этом списке нам будет не 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. В главе 4 мы рассматривали функцию `CONVERT` по отношению к типу данных даты. Здесь она преобразует дату рождения человека к строковому типу данных в нормальном для нас (да и для многих других людей) виде: дд.мм.гггг, который задается последним параметром (104).

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",
```

Получить число строк в таблице, не отображая весь список, можно с помощью функции `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 (средняя зарплата), 1500.00 (минимальная), 56000.00 (максимальная), 346900.00 (сумма всех зарплат сотрудников). 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 применять не рекомендуется. Дело в том, что в процессе работы с таблицей вы по разным причинам можете изменять список выбора. В этом случае номера столбцов могут измениться.

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 проводится 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 чем речь) нужно использовать следующее предложение `ORDER BY`:

```
ORDER BY NAME3, NAME1 DESC;
```

Чтобы устранить повторяющиеся данные в списке, в операторе `SELECT` указывают 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 не будет однофамильцев.

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. для указания, 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
87	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 исключение не возникает, просто результат будет содержать нулевое число 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 тех сотрудников, оклад которых превышает средний оклад по организации. Выполните операторы *примера 8.8*.

Пример 8.8. Выбор сотрудников организации ЗАО "ИнфоТел", оклад которых 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 (СТЕ). Получение кода организации в этом примере реализуют 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 применения обобщенного табличного выражения. В сложных запросах использование СТЕ может повысить скорость выполнения запроса, также наглядность созданного оператора.

8.4.3.2. Использование варианта *LIKE*

В варианте *LIKE* строковое значение должно содержать указанные символы. В этом варианте допустимы шаблонные символы: % означает любое, в том числе и нулевое, число 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" поставим шаблонный символ % (*пример 8.10*).

Пример 8.10. Выбор людей, фамилия которых 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

...

Этот запрос вернет всего 21 строку.

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 "ОВ" должна следовать еще одна любая буква (точнее, любой символ).

Чтобы указать, что фамилия должна начинаться на букву А или Б, нужно ввести:

```
WHERE NAME3 LIKE '[аб]%'
```

Результатом будут шесть строк.

ЗАМЕЧАНИЕ

Нужно 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, оклад которых находится в диапазоне от 12 000 до 23 700. 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 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 /N

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 число значений одного столбца.

Мы можем получить список нескольких штатов США по их кодам. Выполните оператор *примера 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) число 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, задано ключевое слово `DISTINCT` для того, чтобы исключить дублирование кодов в списке. Правда, на результат это никак не влияет.

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, оклад которых не меньше оклада любого сотрудника организации с кодом 16. Результатом будет список, состоящий из одиннадцати записей. Здесь более естественной была бы функция 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, как и в предыдущем примере, 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 пяти странах (таблица `REFCTR`), для которых в базе данных присутствуют сведения об их регионах (таблица `REFREG`).

Здесь также существует альтернативный вариант выборки данных. Можно применить агрегатную функцию `COUNT()`:

```
WHERE (SELECT COUNT(*) FROM REFREG  
       WHERE REFCTR.CODCTR = REFREG.CODCTR) <> 0;
```

8.4.4. Соединение таблиц

Соединение таблиц в операторе `SELECT` – одно из наиболее мощных и элегантных средств реляционных баз данных.

8.4.4.1. Внешнее соединение

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 в выбранные строки добавляются данные (значения указанных столбцов) 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 заработная плата не превышает величины средней заработной платы в этой организации. При этом вместо ничего не значащего кода сотрудника мы путем левого внешнего соединения таблиц добавляем в результат выборки из таблицы `PEOPLE` фамилию, имя и отчество каждого человека, выполнив их конкатенацию.

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) присоединяются данные из таблицы PEOPLE, которые удовлетворяют условию соединения в предложении ON. В нашем случае требуется равенство значения столбца CODPEOPLE из таблицы STAFF (внешний ключ) значению столбца COD из таблицы PEOPLE (первичный ключ), 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 алиас (см. дальше). Конкретно в этом операторе, чтобы избежать двусмысленности, мы обязаны указать уточняющее имя для столбца с именем COD, так как это имя встречается в обеих таблицах.

Коль скоро в таблице людей существует подходящий вычисляемый столбец, содержащий фамилию, имя и отчество, то мы можем записать предыдущий оператор и 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 задавать 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 условии соединения. Однако во внутреннем операторе SELECT недопустимы псевдонимы, указанные во внешнем операторе SELECT, здесь мы всегда задаем полное имя таблицы.

Правое внешнее соединение

Правое внешнее соединение является зеркальным отражением левого внешнего соединения. В нем вначале отбираются все строки правой соединяемой таблицы (здесь она становится главной) на основании условий предложения WHERE. Затем к ним добавляются значения из 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 таблиц сотрудников организации 11 и 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 условию 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 будут 118 строк. Это строки таблицы STAFF плюс строки таблицы PEOPLE минус дублирующие строки.

Двойное соединение

Рассмотрим пример двойного внешнего соединения, т. е. случай, когда к первой таблице присоединяется не одна, а уже две таблицы.

Чтобы при отображении сотрудников организаций видеть 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 мы 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 КЛЮЯ 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.FULLNAME 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 выбираются полное имя человека и полное имя супруга (супруги).

ЗАМЕЧАНИЕ

Обратите внимание на то, что если в других случаях применения оператора SELECT, когда в выборке присутствуют *разные* таблицы, мы иногда можем не использовать псевдонимы таблиц, то в данном случае мы *обязаны* для таблиц задавать псевдонимы, чтобы точно указывать в операторе, к какой именно таблице, к какой роли таблицы в операторе 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.FULLNAME AS "Супруг / супруга",
       PM.FULLNAME AS "Мать",
       PF.FULLNAME 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.

Данные, как вы можете заметить, весьма информативные. На основании такой структуры таблицы людей можно определять практически любые родственные отношения. Будет желание — отыщите племянников, племянниц, братьев, сестер конкретного человека. Я этого пока не делал. А хотелось бы.

8.4.4.2. 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 сотрудников `STAFF` и таблицы людей `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 повлияет.

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 возвращаются 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 АТТО"
38500.00	38500.00	38500.00	4	РΙΑН
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	РΙΑН
22684.61	56000.00	7300.00	13	ЗАО "ИнфоТел"

Основное требование к составу предложения `HAVING` — 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`.

Число 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 применим 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

```

Несколько слов о структуре этой таблицы я скажу немного позже. Выберем данные из 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

```

Мы получим 13 строк.

Цех	Операция	Месяц	Товар	Количество
---	-----	----	-----	-----
NULL	*** Итог ***	NULL	NULL	541.50
01	Расход	NULL	NULL	24.56
01	*** Итог ***	NULL	NULL	541.50
01	Приход	NULL	NULL	516.94
01	Расход	02	NULL	12.56

01	Приход	02	NULL	506.94
01	Приход	02	Goods1	232.60
01	Расход	02	Goods1	12.56
01	Приход	02	Goods2	274.34
01	Приход	03	NULL	10.00
01	Расход	03	NULL	12.00
01	Приход	03	Goods1	10.00
01	Расход	03	Goods1	12.00

Строка, где столбцы имеют значение NULL, является итоговой. Строка, 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 выполнить и для других столбцов, включенных в функцию.

Теперь выполним выборку с помощью 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
```

Результатом будут 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. У нас она служит 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. Поэтому дублированные строки удаляются из результата. Если в оператор UNION добавить ALL, то результат будет содержать шесть строк, где каждый штат отображается дважды.

В следующей *главе 9 (пример 9.5)* показан более осмысленный пример использования оператора UNION в представлении.

В скрипте *примера 8.34* иллюстрируется действие 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 будут две строки, которые возвращаются и первым, и вторым запросом. Это штаты Нью-Йорк и Мериленд.

Что дальше?

В следующей главе мы рассмотрим представления, для получения которых используется оператор SELECT.

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 БД 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 того, представления могут использоваться для сокрытия от пользователя значений некоторых столбцов таблиц, к которым у этого пользователя нет полномочий. При помощи представлений такой пользователь может получить доступ к значениям только тех столбцов таблицы (таблиц), которые ему реально необходимы для работы.

Представления дают возможность 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 индексируемыми. Для представления допустимо создание уникального кластерного индекса. В этом случае результирующий набор данных представления будет физически храниться в БД. 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 БД. Результат получается при объединении данных, выбранных операторами `SELECT`, с помощью операторов `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 приведен в *листинге 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 }
```

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, столбцам присваиваются имена, возвращаемые оператором `SELECT`. Число столбцов в списке и в операторе `SELECT` должно совпадать. 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 объекты, указанные в `SCHEMABINDING`, должны находиться в одной БД.

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`. Следовательно, 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 будут разрешены только такие изменения данных в базовых таблицах представления, 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 предназначен оператор `ALTER VIEW` (листинг 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> ::= { ENCRPTION | SCHEMABINDING | VIEW_METADATA }
```

Оператор в точности повторяет все конструкции оператора создания представления.

9.1.3. Удаление представления

Для удаления группы представлений используется оператор `DROP VIEW` (листинг 9.3).

Листинг 9.3. Синтаксис оператора DROP VIEW

```
<оператор DROP VIEW> ::=  
  DROP VIEW [ <схема>. ] <имя представления>  
  [, [ <схема>. ] <имя представления> ] ... ;
```

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;  
GO  
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. После удаления представления обязательно должна идти команда GO, иначе вы получите ошибку создания представления.

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, задать 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 обычным оператором 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, включающего внешнее соединение

```
USE BestDatabase;
GO
IF OBJECT_ID('VIEW_AREAS', 'V') IS NOT NULL
    DROP VIEW VIEW_AREAS;
GO
CREATE VIEW VIEW_AREAS
    (CODCTR, NAMECTR, CODREG, CODAREA, NAMEAREA)
AS
SELECT A.CODCTR, C.NAME, A.CODREG, A.CODAREA, A.NAMEAREA
    FROM REFAREA A
    LEFT OUTER JOIN REFCTR C
        ON A.CODCTR = C.CODCTR;
GO
```

Отобразите районы всех стран, упорядочив строки по названию района:

```
SELECT * FROM VIEW_AREAS
    ORDER BY NAMEAREA;
GO
```

В полученном списке можно увидеть, сколько районов с одинаковым названием присутствует в разных регионах стран.

Для такого представления нельзя создать индекс, потому что оператор SELECT содержит внешнее соединение. Допустимым в этом случае 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;
GO
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

```

Сейчас создадим представление, в котором используется оператор UNION. В этом представлении осуществляется выборка данных не из одной, а из трех таблиц. Это таблицы, описывающие структуру учебной дисциплины, которая состоит из разделов, тем разделов, уроков тем.

Несколько упрощенная структура таблиц показана в *примере 9.4*.

Пример 9.4. Таблицы учебной дисциплины

```

/**/ Дисциплина /**/
CREATE TABLE SUBJ_SPEC
( CODSUBJ      VARCHAR(20) NOT NULL, /* Имя дисциплины */
  CODSUBJSPEC  VARCHAR(10),          /* Код дисциплины */
  NAMESUBJ     VARCHAR(120),         /* Название дисциплины */
  CONSTRAINT PK_SUBJ_SPEC PRIMARY KEY (CODSUBJ)
);

/**/ Раздел дисциплины специальности /**/
CREATE TABLE SUBJ_SECTION
( CODSUBJ      VARCHAR(20) NOT NULL, /* Имя дисциплины */
  CODSECT      SMALLINT NOT NULL,    /* Номер раздела */
  NAMESECT     VARCHAR(100),         /* Название раздела */
  CONSTRAINT PK_SUBJ_SECTION PRIMARY KEY (CODSUBJ, CODSECT),
  CONSTRAINT FK_SUBJ_SECTION
    FOREIGN KEY (CODSUBJ) REFERENCES SUBJ_SPEC (CODSUBJ)

```

```

        ON DELETE CASCADE
        ON UPDATE CASCADE
    );

    /*** Тема дисциплины ***/
CREATE TABLE SUBJ_TOPIC
( CODSUBJ      VARCHAR(20) NOT NULL, /* Имя дисциплины */
  CODSECT      SMALLINT NOT NULL,    /* Номер раздела */
  CODTOPIC     SMALLINT NOT NULL,    /* Номер темы */
  NAMETOPIC    VARCHAR(120),         /* Название темы */
  CONSTRAINT PK_SUBJ_TOPIC PRIMARY KEY (CODSUBJ, CODSECT, CODTOPIC),
  CONSTRAINT FK_SUBJ_TOPIC FOREIGN KEY (CODSUBJ, CODSECT)
    REFERENCES SUBJ_SECTION (CODSUBJ, CODSECT)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);

    /*** Урок темы ***/
CREATE TABLE SUBJ_LESSON
( CODSUBJ      VARCHAR(20) NOT NULL, /* Имя дисциплины */
  CODSECT      SMALLINT NOT NULL,    /* Номер раздела */
  CODTOPIC     SMALLINT NOT NULL,    /* Номер темы */
  CODESSON     SMALLINT NOT NULL,    /* Номер урока в теме */
  NAMELESSON   VARCHAR(120),         /* Название урока */
  CONSTRAINT PK_SUBJ_LESSON
    PRIMARY KEY (CODSUBJ, CODSECT, CODTOPIC, CODESSON),
  CONSTRAINT FK1_SUBJ_LESSON
    FOREIGN KEY (CODSUBJ, CODSECT, CODTOPIC)
    REFERENCES SUBJ_TOPIC (CODSUBJ, CODSECT, CODTOPIC)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);

```

Следующее представление (*пример 9.5*) позволяет выбрать данные из трех таблиц.

Пример 9.5. Представление выбора элементов учебной дисциплины

```

CREATE VIEW V_SUBJECTLESSONS
(TYPEREC, CODSUBJ, CODSECT, CODTOPIC, CODESSON, TITLE, EL)
AS
    SELECT '1' AS TYPEREC, CODSUBJ, CODSECT, 0 AS CODTOPIC,
        0 AS CODESSON,
        CONCAT('Раздел ', CODSECT, ' ') AS TITLE, NAMESECT AS EL
    FROM SUBJ_SECTION
    UNION
    SELECT '2' AS TYPEREC, CODSUBJ, CODSECT, CODTOPIC, 0 AS CODESSON,
        CONCAT('Тема ', CODTOPIC) AS TITLE, NAMETOPIC AS EL
    FROM SUBJ_TOPIC

```

```
UNION
SELECT '3' AS TYPEREC, CODSUBJ, CODSECT, CODTOPIC, CODESSON,
      CODESSON AS TITLE, NAMELESSON AS EL
FROM SUBJ_LESSON;
```

Выборку данных при использовании этого представления можно выполнить следующим оператором:

```
SELECT * FROM V_SUBJLESSONS WHERE CODSUBJ = 'Базы данных'
ORDER BY CODSECT, CODTOPIC, CODESSON;
```

Здесь выбираются все разделы, темы и уроки дисциплины БД. Полученные данные упорядочиваются таким образом, чтобы была понятна структура конкретной дисциплины.

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 **Обозревателе объектов** раскрыть папку **Базы данных**, папку **BestDatabase**, щелкнуть правой кнопкой мыши по папке **Представления** и в контекстном меню выбрать элемент **Создать представление**. Появится окно добавления таблицы (*рис. 9.1*), которое содержит вкладки выбора таблицы, представления, функций и синонимов. Текущей будет вкладка **Таблицы**.

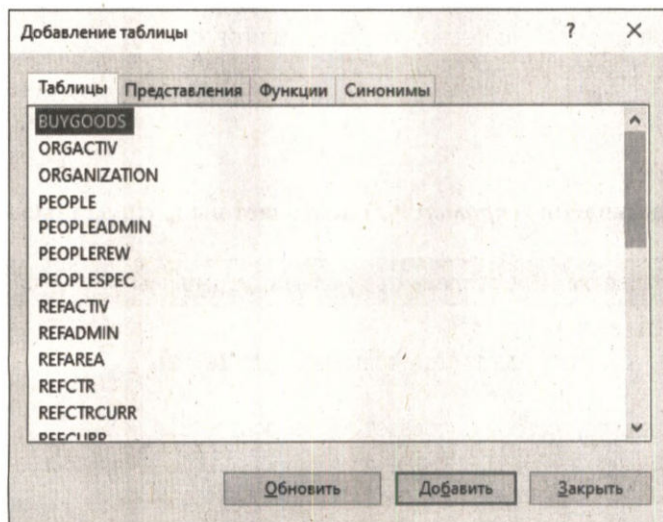


Рис. 9.1. Окно **Добавление таблицы**

Выберите таблицу **REFREG** и щелкните по кнопке **Добавить**, после чего закройте окно. Появится окно выбора столбцов таблицы **REFREG** (*рис. 9.2*).

Отметьте нужные три столбца (рис. 9.3).

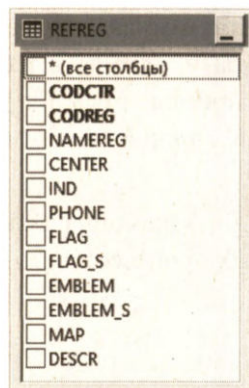


Рис. 9.2. Окно выбора столбцов таблицы

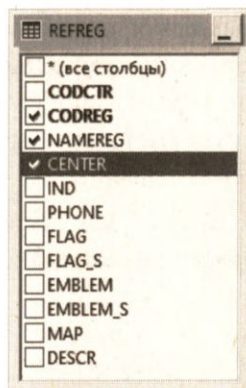


Рис. 9.3. Выбор требуемых столбцов таблицы

В нижней части окна будет сформирован оператор `SELECT`. Туда нужно добавить предложение `WHERE` для отображения только регионов России (рис. 9.4).

Столбец	Псевдо...	Таблица	Выход	Тип сортиро...	Порядок сор...	Фильтр
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 dbo.REFCTR WHERE (NAME = 'РОССИЯ'))))

Рис. 9.4. Созданный оператор `SELECT`

Щелкните правой кнопкой мыши по заголовку формы и выберите **Сохранить**. Появится окно задания имени создаваемого представления (рис. 9.5). Введите имя `VIEW_RUSSIA1` и щелкните по кнопке **ОК**.

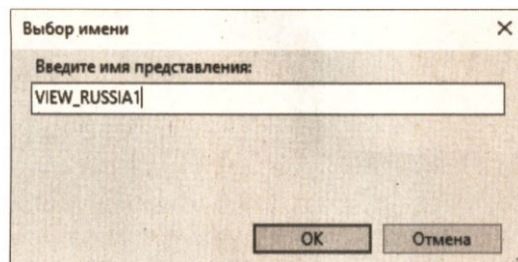


Рис. 9.5. Ввод имени представления

Представление будет сохранено в схеме `dbo` базы данных `BesDatabase`.

Что 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 транзакциям, применяемым в программах, зависит комфортность работы группы пользователей с одной базой данных в архитектуре "клиент / сервер".

Следующую *главу 10* мы посвятим транзакциям, их характеристикам и влиянию этих характеристик на взаимодействие параллельных процессов.

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 БД), 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.

Транзакцию можно объявить явно при помощи оператора `BEGIN TRANSACTION`. Она может быть неявной, когда начинается выполнение оператора работы с БД без оператора старта транзакции.

Транзакции могут быть вложенными.

Внутри транзакции можно создавать точки сохранения. Они позволяют запомнить ситуацию, когда была успешно выполнена группа операций с базой данных. Операции до момента точки сохранения можно отменить.

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 применяется системная переменная `@@TRANCOUNT`, которая содержит число вложенных запущенных на выполнение и не подтвержденных или отмененных в данном соединении с сервером транзакций. Число, большее

единицы, означает, что выполняется транзакция, являющаяся внутренней для основной, главной, транзакции. Для анализа успешности выполнения операторов обращения к базе данных полезна системная переменная @@ERROR. При успешном выполнении операции переменная имеет нулевое значение. @@ERROR очищается при начале выполнения любого другого оператора, поэтому ее следует 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 возможно сокращение TRAN.

10.2. Операторы работы с транзакциями

Оператор BEGIN TRANSACTION (*листинг 10.1*) явно запускает транзакцию.

Листинг 10.1. Синтаксис оператора BEGIN TRANSACTION

```
<оператор BEGIN TRANSACTION> ::=  
  BEGIN TRANSACTION [ <имя> ] [ WITH MARK [ '<описание>' ] ] ;
```

Число символов в имени транзакции не может превышать 32. Имя может задаваться только для самой внешней транзакции в группе вложенных транзакций.

Необязательное предложение WITH MARK указывает, что транзакция отмечается в журнале транзакций. При этом в операторе должно быть задано имя транзакции. Наличие этого предложения позволяет выполнять восстановление базы данных до или после именованной отметки. В предложении WITH MARK можно задать произвольное описание, длина которого не должна превышать 128 символов.

Транзакция подтверждается оператором COMMIT TRANSACTION или отменяется оператором ROLLBACK TRANSACTION.

Синтаксис оператора COMMIT TRANSACTION приведен в *листинге 10.2*.

Листинг 10.2. Синтаксис оператора COMMIT TRANSACTION

```
<оператор COMMIT TRANSACTION> ::=  
  COMMIT TRANSACTION [ <имя> ] ;
```

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 только после подтверждения основной транзакции самого верхнего уровня вложенности.

Аналогично этому оператору применяется оператор COMMIT WORK. Ключевое слово WORK может быть опущено. Различие этих операторов только в отсутствии имени транзакции.

Для отмены транзакции служит оператор `ROLLBACK TRANSACTION` (листинг 10.3).

Листинг 10.3. Синтаксис оператора `ROLLBACK TRANSACTION`

```
<оператор ROLLBACK TRANSACTION> ::=  
    ROLLBACK TRANSACTION [ <имя> | <точка сохранения> ] ;
```

Оператор 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`. В этом случае отменяются лишь изменения, выполненные после создания точки сохранения.

ВНИМАНИЕ!

Независимо от того, на каком уровне вложенности транзакции при выполнении скрипта 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 нуль.

Те же действия по откату транзакции можно выполнить при использовании оператора `ROLLBACK WORK`. Ключевое слово `WORK` может быть опущено.

Для создания точки сохранения предназначен оператор `SAVE TRANSACTION` (листинг 10.4).

Листинг 10.4. Синтаксис оператора `SAVE TRANSACTION`

```
<оператор SAVE TRANSACTION> ::=  
    SAVE TRANSACTION <точка сохранения>;
```

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 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` (листинг 10.5).

Листинг 10.5. Синтаксис оператора `BEGIN DISTRIBUTED TRANSACTION`

```
<оператор BEGIN DISTRIBUTED TRANSACTION> ::=  
    BEGIN DISTRIBUTED TRANSACTION [ <имя> ] ;
```

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 БД. Такие транзакции управляются коорди-

натором распределенных транзакций (Microsoft Distributed Transaction Coordinator, MS DTC). При использовании распределенных транзакций на компьютере должен быть установлен координатор 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.

Задать уровень изоляции, с которым будут выполняться транзакции, позволяет оператор `SET TRANSACTION ISOLATION LEVEL` (листинг 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  
};
```

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 данным, используемым в рамках транзакции.

При описании свойств транзакций употребляется такое понятие, как *фантомные записи*. Этот термин означает, что во время работы какой-то программы при соответствующих уровнях изоляции данного процесса и одновременно с ним выполняющихся параллельных процессах программы могут видеть записи, уже отсутствующие в базе данных на момент выполнения программ.

В SQL Server 2022 поддерживаются пять уровней изоляции:

- ◆ **READ UNCOMMITTED** — неподтвержденное или "грязное" чтение. Этот уровень вообще не изолирует транзакцию от других транзакций. Другие параллельные процессы могут видеть изменения, выполненные, но не подтвержденные в рамках грязной транзакции. 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 много очень серьезных и умных объяснений. Однако на вопрос, применяют ли они этот уровень изоляции, все ответили отрицательно. Лично я такой уровень никогда не использовал в своей практике.

- ◆ **READ COMMITTED** — это значение установлено в системе по умолчанию. При этом уровне изоляции операторы в контексте транзакции видят все подтвержденные изменения в БД, выполненные в других процессах. Неподтвержденные, временные, грязные изменения здесь не видны. База данных представляется как непротиворечивая. Однако данные в БД 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 `READ_COMMITTED_SNAPSHOT` (см. приложение 4):
 - при значении параметра `OFF` (по умолчанию) транзакция не сможет считать строки, которые обновляются другим процессом, до момента подтверждения или отмены транзакции другого процесса. В этом случае возникают блокировки;
 - при значении параметра `ON` текущая транзакция получает мгновенный снимок базы данных на момент старта этой транзакции. Никакие изменения считываемых записей, выполненные другими процессами, не приводят к блокировкам 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 БД. Здесь исключено появление фантомных данных. Чтобы обеспечить такой уровень изоляции, опция `ALLOW_SNAPSHOT_ISOLATION` этой базы данных должна быть установлена в значение `ON` (см. приложение 4).
- ◆ **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 БД других процессов.

Основное правило использования транзакций такое. Независимо от выбранного уровня изоляции рекомендуется транзакции, в которых изменяются данные, делать как можно короче. Транзакции чтения данных, если они не ухудшают возможности параллельных процессов, можно делать достаточно длинными.

Что дальше?

В следующей *главе 11* мы рассмотрим хранимые процедуры и триггеры, которые позволяют предоставить сложные алгоритмы обработки данных на стороне сервера БД и во многих случаях дают возможность уменьшить сетевой трафик.

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 БД и выполняются на стороне сервера, что во многих случаях может сильно сократить сетевой трафик и нагрузку сети.

Как правило, они выполняют какие-то действия с БД, в которой определены, однако это не обязательно. 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 БД — другие хранимые процедуры, функции, триггеры и 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 БД и при изменении метаданных.

Все эти программные компоненты можно создать 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

Для написания хранимых процедур, пользовательских функций и триггеров PSQL создан полноценный язык программирования, который позволяет выполнить произвольную обработку данных — как локальных, так и хранящихся в БД.

В языке допустимы практически любые действия с базой данных, за исключением создания и изменения метаданных. Можно использовать операторы работы с данными БД: INSERT, UPDATE, DELETE и SELECT. 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 представляет собой блок операторов.

11.1.1. 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.

11.1.1.1. Комментарии

В любое место программного кода мы можем поместить комментарий. Текст комментария располагается между символами /* и */, как в большинстве языков программирования. Комментарий может занимать произвольное число строк.

Другой способ задания комментария — два символа "минус": --. В этом случае комментарий продолжается только до конца строки. В некоторых версиях систем (например, в MySQL) после этих символов должен идти пробел.

11.1.1.2. Локальные переменные

Программа может содержать описания локальных переменных, имена которых должны начинаться с символа @.

В теле процедур и триггеров могут присутствовать операторы присваивания, оператор ветвления и операторы циклов.

Вообще нужно сказать, что все эти возможности реализуются 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> ]  
[, <имя переменной> [ AS ] <тип данных> [ = <значение> ]]...;
```

В одном операторе можно объявить произвольное число локальных переменных, разделяя объявления запятыми. При объявлении переменной ей можно также задать начальное значение.

Присвоить значение переменной позволяет оператор SET:

```
SET <имя переменной> = <выражение>;
```

11.1.1.3. Ветвление в программе. Операторы IF и CASE

Ветвление в программе здесь, как и в большинстве языков программирования, осуществляется с помощью оператора IF. Его синтаксис:

```
IF <условие>
{ <оператор> | <блок операторов> }
[ ELSE
{ <оператор> | <блок операторов> } ];
```

Здесь <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 SELECT 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.

ЗАМЕЧАНИЕ

В некоторых языках программирования требуется, чтобы условие в операторе IF было заключено в круглые скобки. Чтобы избежать неприятностей, я во всех языках заключаю условие в скобки.

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 это предложение присутствует.

11.1.1.4. 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 осуществляется соответствующая проверка. Меня это радует.

11.1.1.5. Оператор *GOTO*

Операторы внутри тела процедуры, триггера могут быть снабжены метками. Метка — это идентификатор (имя), после которого следует двоеточие:

```
<метка>: <оператор>;
```

Метки служат 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 применении операторов ветвления и циклов потребность в операторе перехода отпадает.

11.1.1.6. 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 параметр *<целое>* в операторе не указан, 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.

11.1.1.7. Конструкция TRY/CATCH

Можно сказать, что это классическая конструкция, применяемая во многих современных языках программирования для перехвата и обработки ошибок. Эта конструкция недопустима 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 функция 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

```

Результат выполнения этого пакета:

```

NUMBER SEVERITY STATE PROC LINE MESSAGE          @@ERROR
-----
8134    16      1   NULL  2   Обнаружена ошибка: деление на ноль. 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.

11.1.1.8. Оператор *THROW*

Оператор выдает исключение и передает управление блоку `CATCH` в конструкции `TRY/CATCH`. Синтаксис:

```
THROW [ <номер ошибки>, <сообщение>, <состояние> ];
```

Здесь `<номер ошибки>` — это номер ошибки, определенной пользователем. Значение может находиться в диапазоне от 50 000 до 2 147 483 647. Ошибки с номером менее 50 000 являются системными ошибками.

Параметр `<сообщение>` — строка, 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 и клиентские программы. Процедура может получать входные параметры и возвращать выходные параметры. Хранимая процедура выполняется на сервере, что позволяет 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` (листинг 11.1).

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> ::= { ENCRPTION | RECOMPILE |
                        EXECUTE AS <контекст> }
```

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 БД. Имя локальной процедуры должно начинаться с символа #. Глобальная временная процедура видна всем соединениям. Имя такой процедуры должно начинаться с двух символов ##.

Если имя схемы не указано, то хранимая процедура создается в схеме БД по умолчанию.

Для процедуры можно указать параметры. Имя параметра должно начинаться с символа @. Можно объявить до 2100 параметров. Их описания должны отделяться 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 выходным параметром будет 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.

ENCRPTION означает, что система преобразует, кодирует 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. устанавливают, 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 **Обозревателе объектов** базу данных, раскрыв папку **Программирование** и папку **Хранимые процедуры**. Для процедуры можно просмотреть ее параметры (папка **Параметры**), можно вывести ее текст, щелкнув правой кнопкой мыши по имени процедуры и выбрав в меню элементы **Создать скрипт для хранимой процедуры** | **Используя CREATE** | **Новое окно редактора запросов**. 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*).

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> }
```

11.2.3. Удаление хранимой процедуры

Для удаления процедуры предусмотрен оператор `DROP PROCEDURE` (*листинг 11.3*).

Листинг 11.3. Синтаксис оператора `DROP PROCEDURE`

```
DROP PROCEDURE [<имя схемы>.]<имя процедуры>
    [, [<имя схемы>.]<имя процедуры>]... ;
```

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, с помощью которой можно получить уникальное значение, присваиваемое 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 БД значение первичного ключа в таблице `PEOPLE`.

Следует помнить, что создание хранимой процедуры должно быть единственным оператором в пакете. Предыдущая проверка существования в БД 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 присутствует 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 уникальность нам никто не гарантирует. Несколько пользователей могут получить одно и то же значение, что приведет к потере уникальности при попытке поместить в БД новую запись.

В подобном случае лучше задействовать стандартные средства системы, т. е. автоинкрементный ключ. Есть, правда, еще вариант получения уникального целочисленного значения с помощью так называемых последовательностей (`SEQUENCE`).

Сначала в базе данных создается последовательность. Для получения очередного значения используется конструкция `NEXT VALUE FOR`. Для иллюстрации работы с последовательностями 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;
```

Последовательность создается со всеми значениями по умолчанию. При каждом выборе из нее значений мы получаем уникальное число.

Обращение к последовательностям осуществляется вне контекста какой-либо транзакции. Поэтому разные процессы, выполняемые одновременно, получают все равно различные значения.

Теперь рассмотрим более полезную процедуру (*пример 11.5*), которая позволяет получить факториал целого числа ($n!$).

Пример 11.5. Создание хранимой процедуры FACTORIAL

```
USE BestDatabase;
GO
IF OBJECT_ID('FACTORIAL', 'P') IS NOT NULL
    DROP PROCEDURE FACTORIAL;
GO
CREATE PROCEDURE FACTORIAL
    @N INT,
    @NFACTORIAL INT OUTPUT
AS
BEGIN
    DECLARE @I INT = 1;
    SET @NFACTORIAL = 1;
    WHILE @I <= @N
    BEGIN
        SET @NFACTORIAL = @NFACTORIAL * @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 @FACTORIAL AS INT;
EXEC FACTORIAL 5, @FACTORIAL OUTPUT;
PRINT @FACTORIAL;
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 с большими значениями, мы можем выбрать типы данных `BIGINT`, `DECIMAL` и даже `FLOAT`.

Давайте в процедуре зададим проверку на наличие ошибок (превышение допустимого значения исходного числа) с помощью 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('FACTORIAL2', 'P') IS NOT NULL
    DROP PROCEDURE FACTORIAL2;
GO
CREATE PROCEDURE FACTORIAL2
    @N INT,
    @NFACTORIAL INT OUTPUT
AS
BEGIN
    DECLARE @I INT = 1;
    SET @NFACTORIAL = 1;
    BEGIN TRY
        WHILE @I <= @N
        BEGIN
            SET @NFACTORIAL = @NFACTORIAL * @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 @NFACTORIAL = -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: Ошибка арифметического переполнения при преобразовании expression к типу данных int.

-1

Теперь создадим похожую процедуру, которая реализует то же решение, но другим способом (*пример 11.8*).

Пример 11.8. Создание хранимой процедуры FACTORIAL3

```
USE BestDatabase;
GO
IF OBJECT_ID('FACTORIAL3', 'P') IS NOT NULL
```

```
DROP PROCEDURE FACTORIAL3;
GO
CREATE PROCEDURE FACTORIAL3
    @N INT
AS
BEGIN
    DECLARE @I INT = 1;
    DECLARE @NFACTORIAL INT;
    SET @NFACTORIAL = 1;
    BEGIN TRY
        WHILE @I <= @N
        BEGIN
            SET @NFACTORIAL = @NFACTORIAL * @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 @NFACTORIAL = -1;
    END CATCH;
    RETURN @NFACTORIAL;
END;
GO
```

Процедура возвращает значение при помощи оператора RETURN. Обращение к такой процедуре показано в *примере 11.9*.

Пример 11.9. Обращение к процедуре FACTORIAL3

```
USE BestDatabase;
GO
DECLARE @FACTORIAL AS INT;
EXEC @FACTORIAL = FACTORIAL3 5;
PRINT 'Значение = ' + CAST(@FACTORIAL AS VARCHAR(10));
GO
```

Перед именем процедуры в операторе EXECUTE указывается имя переменной, куда будет помещаться возвращаемое функцией значение.

```
EXEC @FACTORIAL = FACTORIAL3 5;
```

Можно создать 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.

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. Функция может использоваться в качестве обычной переменной, значение которой необходимо для вычислений или для отображения результата.

11.3.1. Создание функции

Для создания определенной пользователем функции предусмотрен оператор `CREATE FUNCTION`. Его несколько упрощенный синтаксис приведен в *листинге 11.4*.

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 ]
```

После имени функции (имя должно быть уникальным среди имен функций заданной схемы базы данных) задается список параметров, заключенных в круглые скобки и разделенных запятыми. Если функции не передаются параметры, то скобки 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 не сказываются на начальном значении.

После необязательного ключевого слова WITH указывается список опций функции, которые я не хочу рассматривать в данной книге.

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 (листинг 11.5).

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 ]
```

11.3.3. Удаление функций

Для удаления одной или более функций применяется оператор `DROP FUNCTION`, в котором через запятую перечисляются имена удаляемых функций (листинг 11.6).

Листинг 11.6. Синтаксис оператора `DROP FUNCTION`

```
DROP FUNCTION [<имя схемы>.<имя функции>
[, [<имя схемы>.<имя функции>]... ;
```

11.3.4. Использование функций

Рассмотрим простой пример функции. Это опять факториал целого числа. Создание функции `FACTORIAL7` показано в *примере 11.11*.

Пример 11.11. Создание хранимой процедуры `FACTORIAL7`

```
USE BestDatabase;
GO
IF OBJECT_ID('FACTORIAL7', 'FN') IS NOT NULL
    DROP FUNCTION FACTORIAL7;
GO
CREATE FUNCTION FACTORIAL7
(@N AS INT)
RETURNS FLOAT(8)
AS
BEGIN
    DECLARE @I INT = 1;
    DECLARE @NFACTORIAL FLOAT(8) = 1;
    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, уменьшить сетевой трафик. Кроме того, триггеры могут обеспечить так называемую семантическую целостность данных. Но это оставим за рамками нашей книги.

Напрямую обратиться к триггеру невозможно. Он вызывается автоматически при наступлении соответствующего события БД — 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 и программа, инициировавшая его вызов, выполняются в контексте одной транзакции.

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.

11.4.1. Создание триггеров

Синтаксис оператора создания триггера DML `CREATE TRIGGER` приведен в листинге 11.7.

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
```

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.

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 БД. 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. Здесь важен 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 таблицы, 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 оператором, вызвавшим триггер. Похожим образом таблица `inserted` содержит все строки, добавляемые в основную таблицу. Все подобные таблицы временные 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 далее рассмотрим.

Синтаксис оператора создания триггера DDL приведен в *листинге 11.8*.

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> }
```

После ключевого слова **ON** указывается, какова область действия триггера. Если задано **DATABASE**, то триггер будет срабатывать при наступлении одного из перечисленных далее событий текущей БД. При указании **ALL SERVER** триггер срабатывает при наступлении события для любой БД 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 БД: таблиц, индексов, пользовательских типов данных и многого, многого 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.

Последний тип триггера — триггер входа. Синтаксис создания такого триггера приведен в *листинге 11.9*.

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 приведен в *листингах 11.10—11.12*.

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
```

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> }
```

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> }
```

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*.

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 } ;
```

Листинг 11.15. Синтаксис оператора удаления триггера входа

```
DROP TRIGGER <имя триггера> [, <имя триггера>]...
ON ALL SERVER ;
```

Завершим на этом рассмотрение синтаксических конструкций работы с триггерами. Давайте теперь разберем примеры использования триггеров из реальной жизни.

11.4.4. Использование триггеров

Рассмотрим пару ситуаций, когда триггеры будут действительно полезны.

Первое, огорчительное для меня ограничение состоит в том, что в таблице людей при удалении одной из строк в MS SQL Server нельзя было установить в значение NULL внешние ключи для матери, отца или супруга. Эта нехорошая ситуация (точнее, ошибка системы) может быть разрешена с помощью триггера `INSTEAD OF`.

Оператор создания таблицы людей приведен в *примере 11.13*.

Пример 11.13. Создание таблицы людей

```
CREATE TABLE PEOPLE
( COD          INTEGER IDENTITY(1, 1)
  NAME1        VARCHAR(15),          /* Код человека */
  NAME2        VARCHAR(15),          /* Имя */
  NAME3        VARCHAR(20),          /* Отчество */
  NAME3        VARCHAR(20),          /* Фамилия */
  BIRTHDAY     DATE,                 /* Дата рождения */
  SEX          CHAR(1) DEFAULT '0',  /* Пол */
  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 11.14*.

Пример 11.14. 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.CODOTHERHALF = NULL
        FROM PEOPLE AS P
        INNER JOIN deleted AS D ON P.COD = D.CODOTHERHALF;
    DELETE FROM PEOPLE WHERE COD IN (SELECT COD FROM deleted);
END;
GO
```

Здесь я использую вариант, структуру которого как-то предложил Аарон Бертран (Aaron Bertrand). Это работает, однако немного удивляет использование внутреннего соединения для определения изменяемых строк. Есть более простой вариант:

```
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 CODOTHERHALF = NULL
    WHERE CODOTHERHALF IN (SELECT COD FROM deleted);
```

Рассмотрим еще один пример триггера, который будет создавать таблицу истории окладов сотрудников организаций. Создание таблицы иллюстрирует *пример 11.15*.

Пример 11.15. Оператор создания таблицы истории окладов сотрудников

```
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 11.16*.

Пример 11.16. 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).

– ПРИЛОЖЕНИЯ –

Приложение 1. Двенадцать правил Кодда

Приложение 2. Зарезервированные слова Transact-SQL

Приложение 3. Утилита командной строки sqlcmd

Приложение 4. Характеристики базы данных

Приложение 5. Языки, представленные в SQL Server

Приложение 6. Описание электронного архива

Двенадцать 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 БД 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 БД, 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

Далее приведен 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.

◆ 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

◆ CONSTRAINT	◆ DECIMAL	◆ EXCEPTION
◆ CONSTRAINTS	◆ DECLARE	◆ EXEC
◆ CONSTRUCTOR	◆ DEFAULT	◆ EXECUTE
◆ CONTAINS	◆ DEFERRABLE	◆ FALSE
◆ CONTAINSTABLE	◆ DEFERRED	◆ FILTER
◆ CONTINUE	◆ DELETE	◆ FIRST
◆ CONVERT	◆ DENY	◆ FLOAT
◆ CORR	◆ DEPTH	◆ FOUND
◆ CORRESPONDING	◆ Deref	◆ FREE
◆ COVAR_POP	◆ DESC	◆ FULLTEXTTABLE
◆ COVAR_SAMP	◆ DESCRIBE	◆ FUSION
◆ CREATE	◆ DESCRIPTOR	◆ GENERAL
◆ CROSS	◆ DESTROY	◆ GET
◆ CUBE	◆ DESTRUCTOR	◆ GLOBAL
◆ CUME_DIST	◆ DETERMINISTIC	◆ GO
◆ CURRENT	◆ DIAGNOSTICS	◆ GROUPING
◆ CURRENT_CATALOG	◆ DICTIONARY	◆ HOLD
◆ CURRENT_DATE	◆ DISCONNECT	◆ HOST
◆ CURRENT_DEFAULT_TRANSFORM_GROUP	◆ DISK	◆ HOUR
◆ CURRENT_PATH	◆ DISTINCT	◆ EXISTS
◆ CURRENT_ROLE	◆ DISTRIBUTED	◆ IGNORE
◆ CURRENT_SCHEMA	◆ DOMAIN	◆ IMMEDIATE
◆ CURRENT_TIME	◆ DOUBLE	◆ INDICATOR
◆ CURRENT_TIMESTAMP	◆ DROP	◆ INITIALIZE
◆ CURRENT_TRANSFORM_GROUP_FOR_TYPE	◆ DUMP	◆ EXIT
◆ CURRENT_USER	◆ DYNAMIC	◆ INITIALLY
◆ CURSOR	◆ EACH	◆ EXTERNAL
◆ CYCLE	◆ ELEMENT	◆ FETCH
◆ DATA	◆ ELSE	◆ FILE
◆ DATABASE	◆ END	◆ INOUT
◆ DATE	◆ END-EXEC	◆ INPUT
◆ DAY	◆ EQUALS	◆ FILLFACTOR
◆ DBCC	◆ ERRLVL	◆ FOR
◆ DEALLOCATE	◆ ESCAPE	◆ INT
◆ DEC	◆ EVERY	◆ INTEGER
	◆ EXCEPT	◆ INTERSECTION

◆ INTERVAL	◆ METHOD	◆ OLD
◆ ISOLATION	◆ INDEX	◆ NONCLUSTERED
◆ FOREIGN	◆ INNER	◆ ONLY
◆ FREETEXT	◆ MINUTE	◆ OPERATION
◆ ITERATE	◆ INSERT	◆ NOT
◆ FREETEXTTABLE	◆ MOD	◆ NULL
◆ FROM	◆ MODIFIES	◆ ORDINALITY
◆ LANGUAGE	◆ MODIFY	◆ OUT
◆ LARGE	◆ INTERSECT	◆ NULLIF
◆ LAST	◆ MODULE	◆ OF
◆ LATERAL	◆ MONTH	◆ OVERLAY
◆ LEADING	◆ INTO	◆ OUTPUT
◆ LESS	◆ IS	◆ OFF
◆ FULL	◆ JOIN	◆ OFFSETS
◆ FUNCTION	◆ KEY	◆ PAD
◆ GOTO	◆ MULTISET	◆ PARAMETER
◆ GRANT	◆ NAMES	◆ ON
◆ LEVEL	◆ NATURAL	◆ PARAMETERS
◆ LIKE_REGEX	◆ NCHAR	◆ PARTIAL
◆ LIMIT	◆ KILL	◆ PARTITION
◆ GROUP	◆ LEFT	◆ PATH
◆ LN	◆ NCLOB	◆ POSTFIX
◆ HAVING	◆ NEW	◆ PREORDER
◆ LOCAL	◆ LIKE	◆ PREFIX
◆ LOCALTIME	◆ NEXT	◆ PREPARE
◆ LOCALTIMESTAMP	◆ LINENO	◆ OPEN
◆ LOCATOR	◆ NO	◆ OPENDATASOURCE
◆ HOLDLOCK	◆ NONE	◆ OPENQUERY
◆ IDENTITY	◆ NORMALIZE	◆ PERCENT_RANK
◆ MAP	◆ NUMERIC	◆ OPENROWSET
◆ MATCH	◆ LOAD	◆ OPENXML
◆ IDENTITY_INSERT	◆ MERGE	◆ OPTION
◆ IDENTITYCOL	◆ OBJECT	◆ PERCENTILE_CONT
◆ IF	◆ NATIONAL	◆ PERCENTILE_DISC
◆ IN	◆ NOCHECK	◆ POSITION_REGEX
◆ MEMBER	◆ OCCURRENCES_REGEX	◆ OR

◆ ORDER	◆ ROW	◆ START
◆ PRESERVE	◆ ROWS	◆ STATE
◆ PRIOR	◆ PUBLIC	◆ ROWCOUNT
◆ OUTER	◆ RAISERROR	◆ ROWGUIDCOL
◆ OVER	◆ SAVEPOINT	◆ RULE
◆ PRIVILEGES	◆ SCROLL	◆ SAVE
◆ PERCENT	◆ SCOPE	◆ STATEMENT
◆ RANGE	◆ SEARCH	◆ STATIC
◆ PIVOT	◆ SECOND	◆ SCHEMA
◆ PLAN	◆ READ	◆ SECURITYAUDIT
◆ READS	◆ READTEXT	◆ STDDEV_POP
◆ REAL	◆ SECTION	◆ SELECT
◆ RECURSIVE	◆ RECONFIGURE	◆ STDDEV_SAMP
◆ REF	◆ REFERENCES	◆ STRUCTURE
◆ REFERENCING	◆ SENSITIVE	◆ SUBMULTISET
◆ REGR_AVGX	◆ SEQUENCE	◆ SESSION_USER
◆ REGR_AVGY	◆ SESSION	◆ SUBSTRING_REGEX
◆ REGR_COUNT	◆ SETS	◆ SYMMETRIC
◆ REGR_INTERCEPT	◆ SIMILAR	◆ SET
◆ REGR_R2	◆ SIZE	◆ SETUSER
◆ REGR_SLOPE	◆ REPLICATION	◆ SHUTDOWN
◆ REGR_SXX	◆ RESTORE	◆ SOME
◆ REGR_SXY	◆ RESTRICT	◆ SYSTEM
◆ REGR_SYY	◆ RETURN	◆ TEMPORARY
◆ RELATIVE	◆ SMALLINT	◆ TERMINATE
◆ RELEASE	◆ SPACE	◆ THAN
◆ PRECISION	◆ SPECIFIC	◆ STATISTICS
◆ RESULT	◆ REVERT	◆ SYSTEM_USER
◆ RETURNS	◆ SPECIFICTYPE	◆ TIME
◆ ROLE	◆ REVOKE	◆ TIMESTAMP
◆ ROLLUP	◆ SQL	◆ TABLE
◆ PRIMARY	◆ SQLEXCEPTION	◆ TIMEZONE_HOUR
◆ ROUTINE	◆ SQLSTATE	◆ TABLESAMPLE
◆ PRINT	◆ SQLWARNING	◆ TIMEZONE_MINUTE
◆ PROC	◆ RIGHT	◆ TRAILING
◆ PROCEDURE	◆ ROLLBACK	◆ TRANSLATE_REGEX

◆ TRANSLATION	◆ UNPIVOT	◆ WHEN
◆ TEXTSIZE	◆ VARCHAR	◆ WHERE
◆ THEN	◆ VARIABLE	◆ XMLCONCAT
◆ TREAT	◆ WHENEVER	◆ WHILE
◆ TO	◆ WIDTH_BUCKET	◆ XMLDOCUMENT
◆ TOP	◆ WITHOUT	◆ WITH
◆ TRUE	◆ WITHIN	◆ WRITETEXT
◆ UESCAPE	◆ WINDOW	◆ XMLELEMENT
◆ TRAN	◆ WORK	◆ XMLEXISTS
◆ UNDER	◆ UPDATE	◆ XMLFOREST
◆ UNKNOWN	◆ UPDATETEXT	◆ XMLITERATE
◆ TRANSACTION	◆ USE	◆ XMLNAMESPACES
◆ TRIGGER	◆ WRITE	◆ XMLPARSE
◆ UNNEST	◆ USER	◆ XMLPI
◆ USAGE	◆ VALUES	◆ XMLQUERY
◆ TRUNCATE	◆ VARYING	◆ XMLSERIALIZE
◆ TSEQUAL	◆ XMLAGG	◆ XMLTABLE
◆ USING	◆ XMLATTRIBUTES	◆ XMLTEXT
◆ VALUE	◆ XMLBINARY	◆ XMLVALIDATE
◆ UNION	◆ VIEW	◆ YEAR
◆ UNIQUE	◆ WAITFOR	◆ ZONE
◆ VAR_POP	◆ XMLCAST	
◆ VAR_SAMP	◆ XMLCOMMENT	

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 необязательные. Основные, наиболее часто используемые параметры следующие:

- ◆ -U <имя пользователя> — задает имя пользователя. Вместо 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> или -Z <новый пароль> — задают новый пароль текущего пользователя. Если указан параметр -z, то выполнение утилиты продолжается. В командной строке можно вводить операторы. Если же указан параметр -Z, то выполнение утилиты завершается.

- ◆ `-S <имя сервера>[<имя экземпляра>]` — параметр задает имя сервера в сети и имя экземпляра сервера БД, 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 БД. При указании этого параметра утилита выдает команду `USE <имя базы данных>`. Если параметр не задан, то используется база данных по умолчанию. Если такая БД 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 нужно вводить 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"` или `-Q "запрос к базе данных"` — позволяют 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 точки 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 число раз.

Переменная в операторах задается в виде `$(<имя переменной>)`.

- ◆ `-?` — выводит краткое описание синтаксиса всех параметров, используемых при обращении к утилите `sqlcmd`.

Характеристики базы данных

В этом приложении описаны все характеристики пользовательских БД, которые вы можете использовать в вашей работе и изменять для ваших существующих баз данных. Большинство характеристик присваиваются БД по умолчанию при ее создании. Эти значения вы можете впоследствии изменить с помощью оператора изменения базы данных `ALTER DATABASE` или в компоненте Management Studio. Некоторые характеристики мы рассмотрим достаточно подробно, относительно других — только сообщим об их существовании. Если они вам понадобятся, вы найдете необходимые сведения в любой подходящей литературе.

Характеристик у базы данных очень много. Традиционно их группируют по нескольким категориям.

Не существует нормальной терминологии для названия соответствующих характеристик, свойств, опций. Поэтому я в списке параметров БД привожу ключевые слова, используемые в операторе `ALTER DATABASE` для изменения значения соответствующего параметра, и в скобках названия, отображаемые в окне **Свойства базы данных** на вкладке **Параметры** для свойств БД в диалоговых средствах компонента Management Studio. Окно **Свойства базы данных** появляется, когда в **Обозревателе объектов** вы щелкаете правой кнопкой мыши по имени интересующей вас базы данных и в появившемся контекстном меню выбираете команду **Свойства**. В этом окне в левой части нужно выбрать страницу **Параметры**.

Пример окна **Свойства базы данных** с текущей страницей **Параметры** для БД BestDatabase сразу после ее создания со всеми характеристиками, присваиваемыми базе по умолчанию при ее создании, показан на *рис. П4.1*.

При создании базы данных в языке Transact-SQL с помощью оператора `CREATE DATABASE` вы можете задать только значения для двух параметров — имя БД и порядок сортировки (Collate). Значения всех остальных параметров устанавливаются по умолчанию — выбираются из системной БД model. Имя БД — единственный обязательный параметр при создании новой пользовательской базы данных.

Значения по умолчанию для всех остальных параметров заданы в системной БД model, и некоторые из них пользователь может изменить в процессе работы с системой. Там же указан и порядок сортировки по умолчанию. Он задается во время установки SQL Server. Если вы вносите изменения для значения любого из пара-

метров в БД 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 БД model.

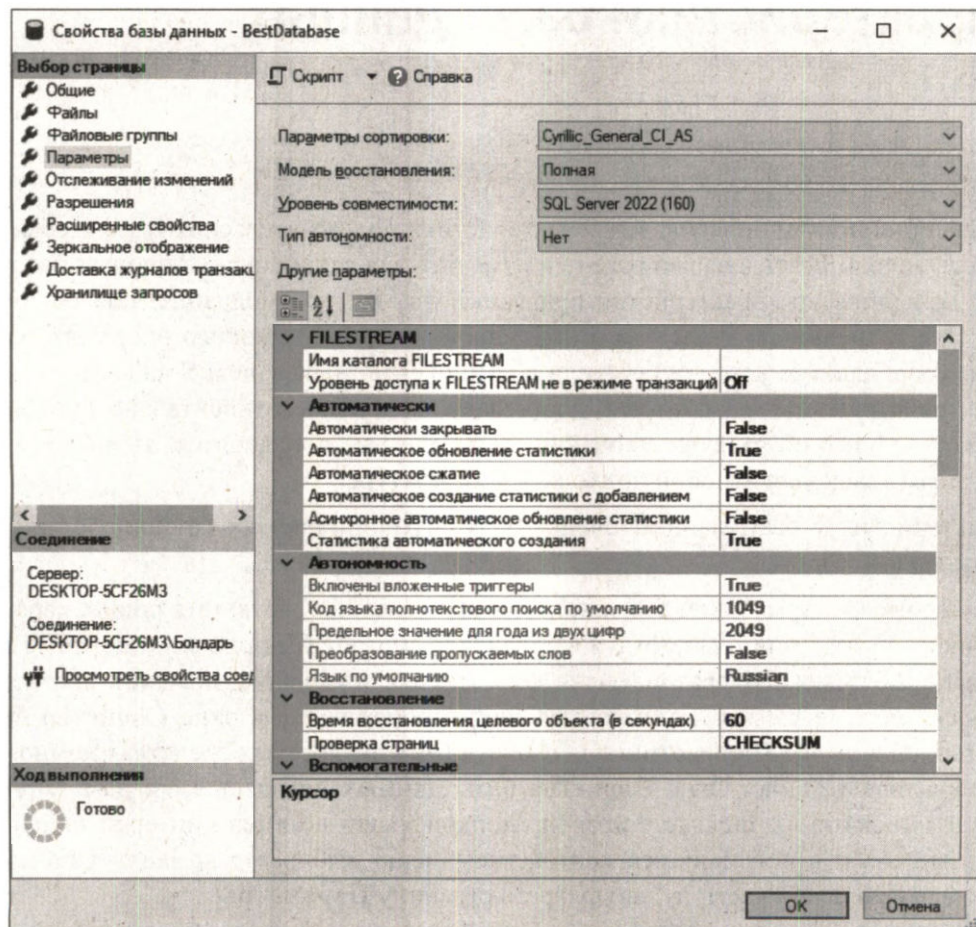


Рис. П4.1. Страница **Параметры** окна **Свойства базы данных** в Management Studio

Для изменения значения допустимого параметра в БД 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 БД model:

```
ALTER DATABASE model
SET COMPATIBILITY_LEVEL = 120;
```

Любой параметр для существующей пользовательской базы данных в дальнейшем может быть изменен в операторе Transact-SQL ALTER DATABASE, а некоторые парамет-

ры, но не все, можно изменить и с использованием диалоговых средств в компоненте Management Studio.

Рассмотрим параметры по их категориям.

П4.1. Параметры *Auto*

В Management Studio эти параметры представлены в группе **Автоматически**.

В категории присутствуют параметры, определяющие автоматический порядок выполнения некоторых действий с базой данных.

◆ **AUTO_CLOSE** (в Management Studio — **Автоматически закрывать**). Автоматическое закрытие базы данных. Можно изменять в БД 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 БД можно копировать, перемещать в другое место, удалять. При последующем подключении любого пользователя к такой базе данных она автоматически открывается для работы;
- если установлено значение **OFF (False)**, то и после отключения последнего пользователя БД остается открытой.

Значение устанавливается в операторе **ALTER DATABASE** заданием варианта **SET AUTO_CLOSE { ON | OFF }** или при помощи выбора в Management Studio в поле **Автоматически закрывать** из выпадающего списка **False** (соответствует **OFF**) или **True** (соответствует **ON**).

◆ **AUTO_CREATE_STATISTICS** (в Management Studio — **Статистика автоматического создания**). Задаёт или отключает автоматическое создание статистики по базе данных, требуемой для оптимизации запроса. Можно изменять в БД 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 статистику можно создать вручную.

Значение устанавливается в операторе **ALTER DATABASE** заданием варианта **SET AUTO_CREATE_STATISTICS { ON | OFF }** или при помощи выбора в Management Studio в поле **Статистика автоматического создания** **False** или **True**.

◆ **AUTO_UPDATE_STATISTICS** (в Management Studio — **Автоматическое обновление статистики**). Задаёт или отключает автоматическое обновление статистики по базе данных, требуемой для оптимизации запроса. Можно изменять в БД 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 **Автоматическое обновление статистики** из выпадающего списка **False** или **True**.

◆ **AUTO_UPDATE_STATISTICS_ASYNC** (в Management Studio — **Асинхронное автоматическое обновление статистики**). Влияет на выполнение запросов, которые требуют обновления статистики по базе данных. Можно изменять в БД model:

- если указано **ON (True)**, то запрос, инициирующий обновление статистических данных, не будет ожидать завершения этого обновления, а будет выполнять оптимизацию на основе устаревшей статистики. Последующие же запросы будут использовать уже обновленные статистические данные;
- если задано **OFF (False)**, то запрос, инициирующий обновление статистических данных, будет ожидать завершения этого обновления. Обновленная статистика будет учтена 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 **Асинхронное автоматическое обновление статистики** из выпадающего списка **False** или **True**. Значение этого параметра **ON** не будет иметь эффекта, если параметр **AUTO_UPDATE_STATISTICS** установлен в **OFF**.

◆ **AUTO_SHRINK** (в Management Studio — **Автоматическое сжатие**). Задаёт или отключает режим автоматического сжатия файлов базы данных. Периодически во время проверок файлы БД могут автоматически сжиматься системой, если в них существует неиспользуемое пространство на внешнем носителе. Можно изменять в БД 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 БД 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 **Автоматическое сжатие** из выпадающего списка **False** или **True**.

П4.2. Параметры доступности базы данных

В Management Studio эти параметры представлены в группе **Состояние**.

В категории присутствуют параметры, определяющие возможность выполнения различных действий с базой данных и ограничения на количество и характеристики пользователей, одновременно подключенных к БД.

◆ **Состояние базы данных.** Некоторые значения можно установить с помощью оператора `ALTER DATABASE`. В Management Studio изменять состояние базы данных нельзя. Нельзя также изменять и в БД model. Состояние может принимать следующие значения:

- **ONLINE** (в Management Studio это состояние отображается как `NORMAL | AUTO_CLOSED`). 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 БД разрешен лишь для членов роли сервера `sysadmin`. Такое состояние обычно применяется для диагностики базы данных;
- **RESTORING**. База данных недоступна. В это состояние БД переводится, когда происходит восстановление файлов данных базы данных;
- **RECOVERING**. База данных недоступна, она находится в процессе восстановления. После завершения восстановления БД автоматически будет переведена в оперативное состояние (`ONLINE`);
- **RECOVERY_PENDING**. База данных недоступна. В процессе восстановления БД 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 БД — `ONLINE`, `OFFLINE` и `EMERGENCY`. Другие состояния устанавливаются системой при появлении конкретных условий или при выполнении некоторых функций с базой данных.

◆ **Возможность изменения данных в БД** (в Management Studio — **База данных доступна только для чтения**). Значение по умолчанию можно изменять в БД model:

- **READ_WRITE (False)**. База данных доступна для чтения и для изменения. Значение по умолчанию;
- **READ_ONLY (True)**. Из БД 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 **База данных доступна только для чтения** из выпадающего списка **False** (соответствует `READ_WRITE`) или **True** (соответствует `READ_ONLY`).

- ◆ Допустимое число пользователей, подключаемых к базе данных (в Management Studio — **Ограничение доступа**). Можно изменять в БД 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. Число пользователей не ограничивается, но к базе данных могут подключаться только пользователи, относящиеся к роли БД 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 **Ограничение доступа** из выпадающего списка соответствующего значения.

П4.3. Параметры автономной базы данных

В Management Studio эти параметры представлены в группе **Включение**.

Определяют характеристики автономной базы данных.

- ◆ DEFAULT_FULLTEXT_LANGUAGE (в Management Studio — **Язык по умолчанию**). Задаёт язык базы данных по умолчанию для полнотекстового поиска в индексированных столбцах.
- ◆ DEFAULT_LANGUAGE (в Management Studio — **Код языка полнотекстового поиска по умолчанию**). Задаёт язык по умолчанию для выполнения полнотекстового поиска в виде кода языка.
- ◆ NESTED_TRIGGERS (в Management Studio — **Включены вложенные триггеры**). Задаёт 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 — **Преобразование пропускаемых слов**). Задаёт поведение сервера БД 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 число 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 — **Предельное значение года из двух цифр**). Задаёт значение года в диапазоне между 1753 и 9999. По умолчанию 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 выбрано столетие, следующее за указанным в операторе столетием.

П4.4. Параметры восстановления

В Management Studio эти параметры представлены в группе **Восстановление**.

В строке **Проверка страниц** можно выбрать следующие варианты:

- ◆ **PAGE_VERIFY** (Проверка страниц в Management Studio). Задаёт способ обнаружения повреждённых страниц базы данных. Можно изменять в БД *model*.
- ◆ **CHECKSUM**. Значение по умолчанию. При помещении изменённой 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 контрольных сумм 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.

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 выпадающего списка соответствующего значения.

Время восстановления целевого объекта в секундах в Management Studio задаёт максимальную границу времени начала восстановления разрушенной базы данных.

П4.5. Общие параметры SQL

В Management Studio эти параметры представлены в группе **Вспомогательные**.

В этой категории больше всего параметров. Это в первую очередь те параметры, которые влияют на поведение системы при выполнении различных операций с метаданными и данными в пользовательской БД: 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 по умолчанию**). 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 БД model:

- **ON (True).** Значение столбца 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 по умолчанию** из выпадающего списка значения **False** или **True**.

- ◆ **ANSI_NULLS (в Management Studio — Включены ANSI NULL).** Задаёт соответствие стандарту ANSI для реляционных баз данных относительно сравнения любых значений с пустым значением NULL. Можно изменять в БД model:

- **ON (True).** Соответствует стандарту ANSI: любое сравнение со значением NULL даёт результат UNKNOWN;
- **OFF (False).** Значение по умолчанию. При сравнении строковых данных не в кодировке Unicode 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 **Включены ANSI NULL** из выпадающего списка значения **False** или **True**.

- ◆ **ANSI_PADDING (в Management Studio — Включено заполнение ANSI).** 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. Изменённое значение этого параметра влияет только на определение столбцов, создаваемых после изменения данного параметра. Можно изменять в БД 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 **Включено заполнение ANSI** из выпадающего списка значения **False** или **True**.

- ◆ **ANSI_WARNINGS (в Management Studio — Включены предупреждения ANSI).** 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 БД 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 **Включены предупреждения ANSI** из выпадающего списка значения **False** или **True**.

- ◆ **ARITHABORT (в Management Studio — Включено арифметическое прерывание).** 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 БД 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 **Включено арифметическое прерывание** из выпадающего списка значения **False** или **True**.

- ◆ **CONCAT_NULL_YIELDS_NULL (в Management Studio — Объединение со значением NULL дает NULL).** Определяет результат конкатенации (соединения) двух строковых данных, когда одно из строковых значений имеет пустое значение NULL. Можно изменять в БД 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 числом 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 **Объединение со значением NULL дает NULL** из выпадающего списка значения **False** или **True**.

- ◆ **DATE_CORRELATION_OPTIMIZATION (в Management Studio — Включена оптимизация корреляции дат).** Определяет поддержание статистики между таблицами, связанными ограничением FOREIGN KEY и содержащими столбцы типа данных datetime. Можно изменять в БД 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 **Включена оптимизация корреляции дат** из выпадающего списка значения **False** или **True**.

◆ **NUMERIC_ROUNDABORT** (в Management Studio — **Прерывание округления**). Задаёт возможность появления ошибки при потере точности в процессе вычисления числового значения. Можно изменять в БД 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 **Прерывание округления** из выпадающего списка значения **False** или **True**.

◆ **PARAMETERIZATION** (в Management Studio — **Параметризация**). Условие выполнения параметризации запросов. Можно изменять в БД model:

- **SIMPLE (Simple)**. Параметризация основывается на поведении базы данных по умолчанию;
- **FORCED (Forced)**. Выполняется параметризация всех запросов в БД.

Значение устанавливается в операторе ALTER DATABASE заданием варианта SET PARAMETERIZATION { SIMPLE | FORCED } или при помощи выбора в Management Studio в поле **Параметризация** из выпадающего списка соответствующего значения.

◆ **QUOTED_IDENTIFIER** (в Management Studio — **Включены идентификаторы в кавычках**). Определяет допустимость использования кавычек ("двойных кавычек") при задании идентификаторов с разделителями. Можно изменять в БД 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 **Включены идентификаторы в кавычках** из выпадающего списка значения **False** или **True**.

◆ **RECURSIVE_TRIGGERS** (в Management Studio — **Включены рекурсивные триггеры**). Определяет допустимость срабатывания рекурсивных триггеров AFTER. Можно изменять в БД 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 **Включены рекурсивные триггеры** из выпадающего списка значения `False` или `True`.

П4.6. Параметры компонента Service Broker

Управляют поведением компонента Service Broker. В базе данных `model` изменить это значение нельзя.

- ◆ `ENABLE_BROKER` (**Включен компонент Broker**, значение `True`). Включает компонент Service Broker, запускает доставку сообщений. В базе данных сохраняется значение идентификатора компонента Service Broker. Значение по умолчанию.
- ◆ `DISABLE_BROKER` (**Включен компонент Broker**, значение `False`). Отключает компонент Service Broker. Отключается доставка сообщений. В базе данных сохраняется значение идентификатора компонента Service Broker.
- ◆ `NEW_BROKER` (**Идентификатор компонента Service Broker**, 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 БД завершаются с ошибками.
- ◆ `HONOR_BROKER_PRIORITY { ON | OFF }` (**Учитывать приоритеты компонента Service Broker**, 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 **Включен компонент Broker**, значение `True`) и `DISABLE_BROKER` (**Broker Enabled**, значение `False`). Значения остальных параметров этой группы могут устанавливаться только в операторе `ALTER DATABASE`.

- 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 Unicode;
- ◆ `alias` — альтернативное название языка (алиас, псевдоним). Например, если официальное название русского языка представлено словом "русский", то альтернативное — словом "Russian". Названия также представлены в кодировке Unicode;
- ◆ `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 следующий код:

```
USE master;
GO
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;
GO
```

Список присутствующих в системе языков можно найти по адресу <http://msdn.microsoft.com/ru-ru/library/ms190303.aspx>. По сравнению с SQL Server 2014 список не изменился.

Описание электронного архива

Электронный архив к книге расположен на FTP-сервере издательства по адресу <https://zip.bhv.ru/9785977518055.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. Пб.1*.

Таблица Пб.1. Структура электронного архива

Файл	Описание
01-Create.sql	Создание базы данных и таблиц
02-Insert1.sql	Добавление в базу данных стран и регионов России, штатов США, графств Великобритании, штатов Австралии, районов России, районов США
03-Insert2.sql	Добавление справочника видов деятельности, справочника товаров
04-Insert3.sql	Добавление организационно-правовых форм, видов организаций, форм собственности, специальностей, административных и уголовных правонарушений. Добавление людей, включая сведения о родственных связях. Добавление организаций, сотрудников организаций и правонарушений людей

Предметный указатель

1

12 правил Кодда, 33

D

Data Definition Language, DDL, 29, 52

O

Online Analytical Processing, OLAP, 41

Online Transaction Processing, OLTP, 41

A

Автономные базы данных (contained), 135

B

Владелец базы данных, 62

Выборка данных, 415

- использование операторов сравнения, 429, 431, 432, 433, 434
- функции ANY и SOME, 435
- функция ALL, 435
- функция EXISTS, 436
- функция SINGULAR, 436

Вычисляемые столбцы, 274

Г

Группировка результатов выборки, 444

Д

Декларативная целостность, 29

Денормализация таблиц, 41

Добавление данных — оператор INSERT, 396

Добавление, изменение или удаление строк таблицы — оператор MERGE, 408

З

Задание уровня изоляции транзакции — оператор SET TRANSACTION ISOLATION LEVEL, 464

Запуск и останов экземпляра сервера, 56

Запуск распределенной транзакции — оператор BEGIN DISTRIBUTED TRANSACTION, 463

Запуск транзакции — оператор BEGIN TRANSACTION, 462

Значение NULL, 26

И

Изменение базы данных

- диалоговые средства Management Studio, 128

- оператор ALTER DATABASE, 116

Изменение данных — оператор UPDATE, 403

Изменение индекса

- диалоговые средства Management Studio, 394

- оператор ALTER INDEX, 388

Изменение представления

- оператор ALTER VIEW, 453

Изменение схемы базы данных

- оператор ALTER SCHEMA, 144

Изменение таблицы

- диалоговые средства Management Studio, 341
- оператор ALTER TABLE, 336

Изменение триггера — оператор ALTER TRIGGER, 485

Изменение функции, определенной пользователем, — оператор ALTER FUNCTION, 481

Изменение хранимой процедуры — оператор ALTER PROCEDURE, 474

Индексы, 27, 270, 369

К

Кластерный индекс, 373

Ключи в таблицах, 27

- внешний ключ, 28
- задание первичных ключей, 31
- первичный ключ, 27
- уникальный ключ, 28

Конструкция TRY/CATCH, 471

Копирование и восстановление баз данных, 148

- диалоговые средства Management Studio, 149
- оператор BACKUP, 148
- оператор RESTORE, 149

Л

Локальные переменные, 158

Н

Нормализация таблиц, 36

- вторая нормальная форма, 39
- нормальная форма Бойса–Кодда, 40
- первая нормальная форма, 37
- пятая нормальная форма, 40
- третья нормальная форма, 39
- четвертая нормальная форма, 40

О

Обобщенное табличное выражение, 395

Ограничение

- CHECK, 273
- внешнего ключа, 271

- первичного ключа, 269

- уникального ключа, 269

Ограничения таблицы, 29, 268

Оператор

- CREATE TRIGGER, 483
- GOTO, 470
- RETURN, 470
- THROW, 472

Операторы

- ALTER DATABASE, 116
- ALTER FUNCTION, 481
- ALTER INDEX, 388
- ALTER PROCEDURE, 474
- ALTER SCHEMA, 144
- ALTER TABLE, 336
- ALTER TRIGGER, 485
- ALTER VIEW, 453
- BACKUP, 148
- BEGIN DISTRIBUTED TRANSACTION, 463
- BEGIN TRANSACTION, 462
- COMMIT TRANSACTION, 462
- CREATE COLUMNSTORE INDEX, 379
- CREATE DATABASE, 72
- CREATE FUNCTION, 480
- CREATE INDEX, 371
- CREATE PROCEDURE, 472
- CREATE SPATIAL INDEX, 385
- CREATE TABLE, 262
- CREATE TRIGGER, 483
- CREATE SCHEMA, 143
- CREATE VIEW, 452
- CREATE XML INDEX, 380
- DELETE, 407
- DROP DATABASE, 81
- DROP FUNCTION, 482
- DROP INDEX, 387
- DROP PROCEDURE, 474
- DROP SCHEMA, 144
- DROP TABLE, 333
- DROP TRIGGER, 486
- DROP VIEW, 453
- EXCEPT, 424
- INSERT, 396
- INTERSECT, 424
- MERGE, 409
- RESTORE, 149
- ROLLBACK TRANSACTION, 463
- SAVE TRANSACTION, 463
- SELECT, 415

- SET TRANSACTION ISOLATION LEVEL, 464
 - TRUNCATE TABLE, 408
 - UNION, 424
 - UPDATE, 403
- Операции сравнения, 420
- Определение зависимостей таблицы, 330
- Организация циклов. Оператор WHILE, 470
- Отмена (откат) транзакции — оператор ROLLBACK TRANSACTION, 463
- Отношения между таблицами, 29, 34
- "многие ко многим", 35
 - "один к одному", 34
 - "один ко многим", 35

П

- Подтверждение транзакции — оператор COMMIT TRANSACTION, 462
- Пользовательские базы данных, 58
- Порядок сортировки, 25, 62
- Представление, 29, 451, 452

Р

- Разреженные столбцы, 267
- Реляционные базы данных, 23

С

- Секционированные таблицы, 289
- Семантика, 43
- Синтаксис, 43
- Системные базы данных, 58
- БД master, 59
 - БД model, 59
 - БД msdb, 60
 - БД resource, 60
 - БД tempdb, 60
- Системные представления, 70
- Системные хранимые процедуры, 71
- Соединение таблиц, 436
- внутреннее соединение, 443
 - двойное соединение, 440
 - левое внешнее соединение, 437
 - полное внешнее соединение, 439
 - правое внешнее соединение, 439
 - рефлексивное соединение, 442
- Создание columnstore индекса
- оператор CREATE COLUMNSTORE INDEX, 379

- Создание базы данных, 72
- диалоговые средства Management Studio, 112
 - оператор CREATE DATABASE, 72
- Создание индекса
- диалоговые средства Management Studio, 390
 - оператор CREATE INDEX, 371
- Создание индекса XML, 380
- Создание мгновенных снимков базы данных, 141
- Создание определенной пользователем функции — оператор CREATE FUNCTION, 480
- Создание представления
- диалоговые средства Management Studio, 458
 - оператор CREATE VIEW, 452
- Создание пространственного индекса
- оператор CREATE SPATIAL INDEX, 385
- Создание схемы базы данных
- оператор CREATE SCHEMA, 143
- Создание таблицы
- диалоговые средства Management Studio, 307
 - оператор CREATE TABLE, 262
- Создание точки сохранения транзакции — оператор SAVE TRANSACTION, 463
- Создание хранимой процедуры — оператор CREATE PROCEDURE, 472
- Строковые функции, 178
- Схемы базы данных, 143

T

- Таблица, 23, 261
- Таблицы истинности, 419
- Тип данных, 24, 156
- CURSOR, 230
 - HIERARCHYID, 224
 - SQL_VARIANT, 220
 - UNIQUEIDENTIFIER, 229
 - XML, 237
- Типы данных
- даты и времени, 189
 - двоичные, 201
 - пользовательские, 249
 - пространственные, 202
 - символьные, 175
 - числовые, 159
- Транзакция 32, 461
- Триггер, 30, 483

У

Удаление функций — оператор DROP FUNCTION, 482

Удаление базы данных

- диалоговые средства Management Studio, 135

- оператор DROP DATABASE, 81

Удаление данных — оператор DELETE, 407

Удаление индекса

- диалоговые средства Management Studio, 394

- оператор DROP INDEX, 387

Удаление представления

- оператор DROP VIEW, 453

Удаление строк таблицы — оператор TRUNCATE TABLE, 408

Удаление схемы базы данных

- оператор DROP SCHEMA, 144

Удаление таблицы

- диалоговые средства Management Studio, 333

- оператор DROP TABLE, 333

Удаление триггера — оператор DROP TRIGGER, 486

Удаление хранимой процедуры — оператор DROP PROCEDURE, 474

Упорядочение результата — оператор ORDER BY, 427

Уровни изоляции транзакции, 464

Условие выборки, 418

Ф

Файловые группы, 80, 97

Файловые потоки, 324

Функции

- агрегатные, 170
- даты и времени, 194
- логарифмические, 172
- определенные пользователем, 30, 480
- системные, 71
- тригонометрические, 171

Х

Характеристики базы данных, 62

Характеристики файлов базы данных, 64

Хранимые процедуры, 30, 472

Я

Язык

- Transact-SQL, 43
- хранимых процедур и триггеров, 467

Отдел оптовых поставок:

e-mail: opt@bhv.ru



- Полностью обновленные рецепты, учитывающие более широкое использование оконных функций в SQL-приложениях
- Дополнительные примеры, показывающие обширное использование обобщенных табличных выражений (ОТВ) для создания более удобочитаемых и простых решений
- Новые решения, которые делают SQL более полезным для пользователей, не являющихся экспертами в области СУБД, включая специалистов по работе с данными
- Расширенные выражения для обработки чисел и строк

Рассмотрены готовые рецепты для решения практических задач при работе с СУБД Oracle, DB2, SQL Server, MySQL и PostgreSQL. Описаны извлечение записей из таблиц, сортировка результатов запросов, принципы работы с несколькими таблицами, обработка запросов с метаданными. Рассказывается о способах поиска данных средствами SQL, о составлении отчетов и форматировании результирующих множеств, работе с иерархическими запросами. Рассматривается использование оконных функций, обобщенных табличных выражений (ОТВ), сбор данных в блоки, формирование гистограмм, текущих сумм и подсумм, агрегация скользящего диапазона значений. Описан обход строки и ее синтаксический разбор на символы, приведены способы упрощения вычислений внутри строки. Во втором издании учтены все изменения в синтаксисе и архитектуре актуальных реализаций SQL.

Молинаро Энтони, специалист по данным в компании Johnson & Johnson. Занимается исследованиями непараметрических методов, анализом временных рядов и характеристик крупномасштабных баз данных, а также их преобразованием.

де Грааф Роберт, занимает должность главного специалиста по данным в компании RightShip.

Отдел оптовых поставок:

e-mail: opt@bhv.ru

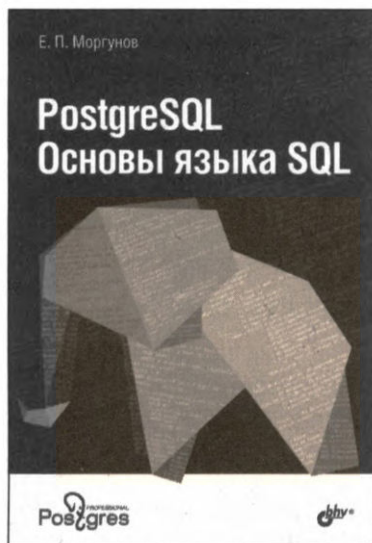


- Узнайте о ключевых шагах по подготовке данных к анализу
- Выполняйте анализ временных рядов
- Используйте когортный анализ для изучения изменений групп с течением времени
- Научитесь применять мощные функции SQL для анализа текста
- Выявляйте отклонения и аномалии в данных
- Установите причинно-следственную связь с помощью анализа экспериментов и результатов А/В-тестирования
- Применяйте SQL для оценки эффективности воронки продаж

Рассказывается о возможностях SQL применительно к анализу данных. Сравниваются различные типы баз данных, описаны методы подготовки данных для анализа. Рассказано о типах данных, структуре SQL-запросов, профилировании, структурировании и очистке данных. Описаны методы анализа временных рядов, трендов, приведены примеры анализа данных с учетом сезонности. Отдельные главы посвящены когортному анализу, текстовому анализу, выявлению и обработке аномалий, анализу результатов экспериментов и А/В-тестирования. Описано создание сложных наборов данных, комбинирование методов анализа. Приведены практические примеры анализа воронки продаж и потребительской корзины.

Танимура Кэти, более 20 лет занимается анализом данных в самых разных отраслях, от финансов до программного обеспечения и сферы потребительских услуг. Кэти управляла командами специалистов по анализу данных в нескольких ведущих технологических компаниях. Имеет богатый опыт работы со стандартом SQL, включая наиболее популярные проприетарные базы данных и базы данных с открытым исходным кодом.

Отдел оптовых поставок:
e-mail: opt@bhv.ru



Учебно-практическое пособие охватывает первую, базовую, часть учебного курса по языку SQL, созданного при участии российской компании Postgres Professional. Учебный материал излагается в расчете на использование системы управления базами данных PostgreSQL. Рассмотрено создание рабочей среды, описаны язык определения данных и основные операции выборки и изменения данных. Показаны примеры использования транзакций, уделено внимание методам оптимизации запросов. Материал сопровождается многочисленными практическими примерами. Пособие может использоваться как для самостоятельного обучения, так и проведения занятий под руководством преподавателя.

Моргунов Евгений, кандидат технических наук, доцент кафедры информатики и вычислительной техники Сибирского государственного университета науки и технологий имени академика М. Ф. Решетнева. До начала преподавательской деятельности более десяти лет проработал программистом и разработчиком баз данных. Имеет более чем двадцатилетний опыт обучения студентов компьютерным дисциплинам. Являясь сторонником использования открытого и свободного программного обеспечения, применяет в преподавании СУБД PostgreSQL. Член Международного общества по инженерной педагогике (IGIP) с 2004 года.

Отдел оптовых поставок:

e-mail: opt@bhv.ru



Вы узнаете, как:

- Начинать работу с реляционной СУБД MySQL и управлять данными
- Развертывать базы данных MySQL на «голом железе», на виртуальных машинах и в облаке
- Конструировать инфраструктуры базы данных
- Кодировать высокоэффективные запросы
- Обеспечивать мониторинг и устранять неполадки баз данных MySQL
- Выполнять эффективные операции резервного копирования и восстановления
- Оптимизировать издержки баз данных в облаке
- Понимать концепции баз данных, в особенности те, которые относятся к MySQL

Книга знакомит с MySQL — самой популярной системой управления базами данных с открытым исходным кодом. Изложены основы MySQL: установка, моделирование и конструирование баз данных, команды SQL и создание новой базы данных. Рассмотрены практические вопросы работы с MySQL: расширенные запросы, транзакции и замковый механизм, проверка эффективности запросов, управление пользователями и привилегиями, использование файлов опций, резервное копирование и восстановление, конфигурирование и настройка сервера. Отдельное внимание уделено мониторингу серверов MySQL, асинхронной и синхронной репликации, кластерным решениям, работе в облаке, балансировке нагрузки и другим продвинутым методам и инструментам.

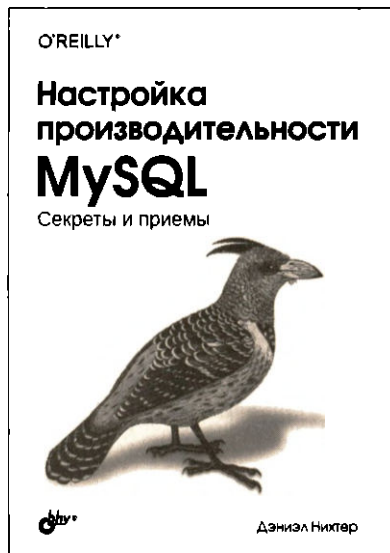
Электронный архив на сайте издательства содержит цветные иллюстрации к книге.

Гриппа Винисиус, старший инженер службы поддержки компаний Percona и ACE Oracle. Помог клиентам Percona разобраться в сотнях различных случаев использования MySQL.

Кузьмичев Сергей, в настоящее время старший инженер технической поддержки в компании Percona, до этого почти десять лет работал администратором баз данных и инженером DevOps.

Отдел оптовых поставок:

e-mail: opt@bhv.ru



- Подробный разбор запросов MySQL и скорости их выполнения
- Агрегация и анализ метрик, формирование отчетов по ним
- Изучение транзакций и блокировок
- Масштабирование, шардирование, репликация MySQL
- Переход в облако

Книга посвящена практическим аспектам работы с MySQL. Рассмотрены приемы сегментирования баз данных, репликации, шардирования. Уделено внимание упорядочиванию транзакций, резервному копированию и бесшовному взаимодействию между предприятием и облаком, что способствует сохранению данных и их эксплуатационной надежности. Также исследованы разнообразные аналитические и мониторинговые инструменты и предложены проверенные методы, актуальные при развитии и долгосрочной поддержке MySQL и других РСУБД.

Нихтер Дэниэл, архитектор баз данных, более 15 лет работает с MySQL. Увлёкся оптимизацией производительности MySQL еще в 2004 году, работая в дата-центре. Из его заметок сложился блог HackMySQL.com, в котором он делился подробностями устройства MySQL и ее инструментария. Затем 8 лет работал в компании Percona, где продолжал разрабатывать инструменты для оптимизации баз данных. Сегодня его инструменты де-факто служат эталоном для компаний во всем мире. Обладатель премии MySQL Community Award, выступает на конференциях, активно участвует в движении Open Source.



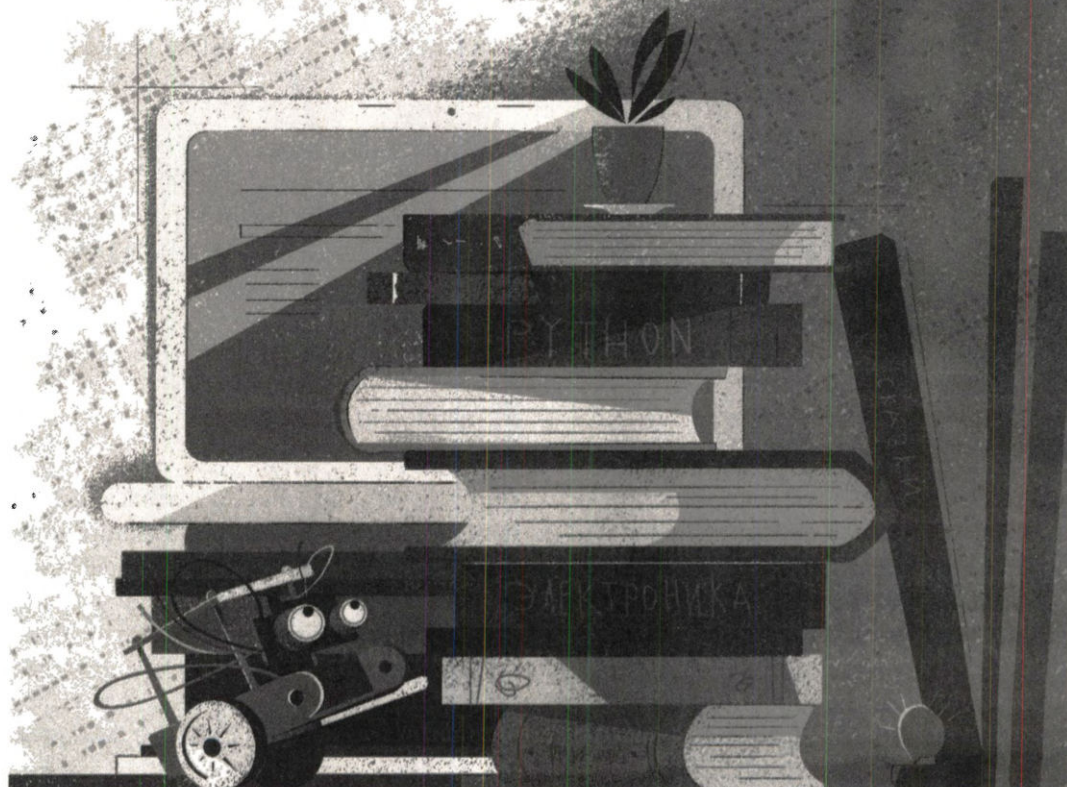
ИНТЕРНЕТ-МАГАЗИН

BHV.RU

КНИГИ, РОБОТЫ,
ЭЛЕКТРОНИКА

Интернет-магазин издательства «БХВ»

- Более 30 лет на российском рынке
- Книги и наборы по электронике и робототехнике по издательским ценам
- Электронные архивы книг и компакт-дисков
- Ответы на вопросы читателей



Интернет-магазин БХВ-Электроника
Скоро открытие!

