

# Microsoft SQL Server 2012

# ОСНОВЫ T-SQL





**МИРОВОЙ  
КОМПЬЮТЕРНЫЙ  
БЕСТСЕЛЛЕР**

Itzik Ben-Gan

Microsoft SQL Server 2012  
T-SQL  
Fundamentals

***Microsoft***<sup>®</sup>

Ицик Бен-Ган

Microsoft SQL Server 2012

# ОСНОВЫ T-SQL



ЭКСМО  
МОСКВА  
2015

УДК 004.45  
ББК 32.973-018.2  
Б 46

Authorized Russian translation of the English edition of Microsoft SQL Server 2012 T-SQL Fundamentals (ISBN 9780735658141) © 2012 Itzik Ben-Gan. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

**Бен-Ган, Ицик.**

Б 46 Microsoft SQL Server 2012. Основы T-SQL / Ицик Бен-Ган ; [пер. с англ. М. А. Райтмана]. — Москва : Эксмо, 2015. — 400 с. — (Мировой компьютерный бестселлер).

ISBN 978-5-699-73617-1

В книге изложены основы программирования на языке T-SQL. Вы научитесь программировать и писать запросы для Microsoft SQL Server 2012, а большое количество примеров и упражнений помогут вам начать создавать эффективный код.

УДК 004.45  
ББК 32.973-018.2

Производственно-практическое издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

**Ицик Бен-Ган**

**MICROSOFT SQL SERVER 2012**

**Основы T-SQL**

(орыс тілінде)

Директор редакции *Е. Капъёв*  
Ответственный редактор *В. Обручев*  
Художественный редактор *Г. Федотов*

ООО «Издательство «Эксмо»  
123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86, 8 (495) 956-39-21.  
Home page: [www.eksmo.ru](http://www.eksmo.ru) E-mail: [info@eksmo.ru](mailto:info@eksmo.ru)  
Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Зорге көшесі, 1 үй.  
Тел. 8 (495) 411-68-86, 8 (495) 956-39-21  
Home page: [www.eksmo.ru](http://www.eksmo.ru) E-mail: [info@eksmo.ru](mailto:info@eksmo.ru)  
Тауар белгісі: «Эксмо»  
Қазақстан Республикасында дистрибьютор және өнім бойынша  
арыз-талаптарды қабылдаушының  
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3-а., литер Б, офис 1.  
Тел.: 8 (727) 2 51 59 89,90,91,92, факс: 8 (727) 251 58 12 вн. 107; E-mail: [RDC-Almaty@eksmo.kz](mailto:RDC-Almaty@eksmo.kz)  
Өнімнің жарамдылық мерзімі шектелмеген.  
Сертификация туралы ақпарат сайты: [www.eksmo.ru/certification](http://www.eksmo.ru/certification)

Сведения о подтверждении соответствия издания согласно законодательству РФ  
о техническом регулировании можно получить по адресу: <http://eksmo.ru/certification/>  
Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 20.12.2014. Формат 70х100<sup>1</sup>/<sub>16</sub>.

Печать офсетная. Усл. печ. л. 32,41.

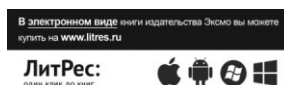
Тираж экз. Заказ

ISBN 978-5-699-73617-1



9 785699 736171 >

ISBN 978-5-699-73617-1



© Райтман М.А., перевод на русский язык, 2015  
© Оформление. ООО «Издательство «Эксмо», 2015

# КРАТКОЕ ОГЛАВЛЕНИЕ

Предисловие.....	12
Введение.....	14
Глава 1. Общие сведения о создании запросов и программировании на языке T-SQL .....	18
Глава 2. Запросы к одиночным таблицам.....	43
Глава 3. Соединения .....	111
Глава 4. Вложенные запросы .....	140
Глава 5. Табличные выражения.....	165
Глава 6. Операторы работы с наборами.....	198
Глава 7. Продвинутые запросы .....	217
Глава 8. Изменение данных .....	251
Глава 9. Транзакции и параллелизм .....	298
Глава 10. Программируемые объекты .....	339
Приложение. Приступаем к работе .....	374
Алфавитный указатель .....	395

# ОГЛАВЛЕНИЕ

<b>Предисловие.....</b>	<b>12</b>
<b>Введение .....</b>	<b>14</b>
Для кого предназначена книга .....	14
Кому книга противопоказана .....	15
Как устроена книга.....	15
Системные требования .....	16
Примеры кода .....	16
Благодарности.....	16
<b>Глава 1. Общие сведения о создании запросов и программировании     на языке T-SQL .....</b>	<b>18</b>
Теоретические основы .....	18
SQL .....	19
Теория множеств .....	20
Исчисление предикатов .....	21
Реляционная модель.....	21
Жизненный цикл данных .....	27
Архитектура SQL Server.....	30
Разновидности SQL Server.....	30
Экземпляры SQL Server.....	32
Базы данных .....	33
Схемы и объекты.....	35
Создание таблиц и обеспечение целостности данных.....	36
Создание таблиц .....	37
Обеспечение целостности данных.....	38
В заключение.....	42
<b>Глава 2. Запросы к одиночным таблицам.....</b>	<b>43</b>
Элементы инструкции SELECT.....	43
Инструкция FROM.....	45
Инструкция WHERE.....	46
Инструкция GROUP BY .....	48
Инструкция HAVING.....	51

---

Инструкция SELECT .....	52
Инструкция ORDER BY .....	57
Фильтры TOP и OFFSET-FETCH .....	59
Краткий обзор оконных функций .....	63
Предикаты и операторы .....	65
Выражение CASE .....	67
Отметки NULL .....	70
Одновременные операции .....	74
Работа с символьными данными .....	75
Типы данных .....	75
Параметры сравнения .....	76
Операторы и функции .....	78
Предикат LIKE .....	84
Работа с датой и временем .....	87
Типы данных даты и времени .....	87
Строковые литералы .....	88
Раздельная работа с датой и временем .....	91
Фильтрация дат по диапазону .....	92
Функции даты и времени .....	93
Извлечение метаданных .....	100
Представления каталога .....	101
Представления информационной схемы .....	102
Системные хранимые процедуры и функции .....	102
В заключение .....	103
Упражнения .....	104
Решения .....	107
<b>Глава 3. Соединения .....</b>	<b>111</b>
Перекрестные соединения .....	111
Синтаксис ANSI SQL-92 .....	112
Синтаксис ANSI SQL-89 .....	113
Перекрестные самосоединения .....	113
Создание числовых таблиц .....	114
Внутренние соединения .....	115
Синтаксис ANSI SQL-92 .....	115
Синтаксис ANSI SQL-89 .....	116
Безопасность внутренних соединений .....	117
Другие разновидности соединений .....	118
Составные соединения .....	118
Не-эквисоединения .....	119
Множественные соединения .....	121
Внешние соединения .....	122
Общие сведения .....	122
Дополнительные сведения .....	125
В заключение .....	131
Упражнения .....	131
Решения .....	136

---



---

<b>Глава 4. Вложенные запросы</b> .....	140
Автономные вложенные запросы .....	140
Примеры со скалярным результатом .....	140
Примеры с множественными значениями .....	142
Коррелирующие вложенные запросы .....	146
Предикат EXISTS .....	148
Примеры сложных вложенных запросов .....	150
Возвращение предыдущих или следующих значений .....	150
Использование текущих агрегатов .....	151
Решение проблем .....	152
В заключение .....	157
Упражнения .....	157
Решения .....	161
<b>Глава 5. Табличные выражения</b> .....	165
Производные таблицы .....	165
Назначение псевдонимов для столбцов .....	167
Использование аргументов .....	168
Вложенность .....	169
Множественные ссылки .....	170
Обобщенные табличные выражения .....	171
Псевдонимы столбцов .....	171
Использование аргументов .....	172
Определение нескольких ОТВ .....	172
Множественные ссылки .....	173
Рекурсивные ОТВ .....	174
Представления .....	176
Внструкция ORDER BY .....	177
Параметры .....	180
Встроенные функции с табличными значениями .....	184
Оператор APPLY .....	185
В заключение .....	188
Упражнения .....	189
Решения .....	193
<b>Глава 6. Операторы работы с наборами</b> .....	198
Объединение .....	199
Мультимножества (оператор UNION ALL) .....	199
Множества (оператор UNION) .....	200
Пересечение .....	201
Множества (оператор INTERSECT) .....	201
Мультимножества (оператор INTERSECT ALL) .....	202
Разность .....	204
Множества (оператор EXCEPT) .....	205
Мультимножества (оператор EXCEPT ALL) .....	206

---

---

Приоритет.....	207
Эмуляция неподдерживаемых логических этапов .....	208
В заключение.....	210
Упражнения.....	210
Решения .....	214
<b>Глава 7. Продвинутое запросы .....</b>	<b>217</b>
Оконные функции.....	217
Ранжирующие функции.....	220
Функции со смещением .....	223
Агрегатные функции .....	225
Разворачивание данных .....	228
Стандартные средства.....	230
Встроенный оператор PIVOT .....	231
Отмена разворачивания данных.....	233
Стандартные средства.....	235
Встроенный оператор UNPIVOT .....	237
Группирующие наборы.....	238
Вложенная инструкция GROUPING SETS.....	239
Вложенная инструкция CUBE .....	240
Вложенная инструкция ROLLUP .....	240
Функции GROUPING и GROUPING_ID.....	241
В заключение.....	244
Упражнения.....	244
Решения .....	247
<b>Глава 8. Изменение данных .....</b>	<b>251</b>
Добавление данных.....	251
Команда INSERT VALUES.....	251
Команда INSERT SELECT .....	253
Команда INSERT EXEC .....	253
Команда SELECT INTO.....	254
Команда BULK INSERT .....	256
Свойство identity и объект последовательности .....	256
Удаление данных .....	265
Команда DELETE.....	266
Команда TRUNCATE .....	266
Команда DELETE в сочетании с оператором JOIN .....	267
Обновление данных .....	268
Команда UPDATE.....	269
Команда UPDATE в сочетании с оператором JOIN .....	270
Обновление с присваиванием .....	272
Слияние данных.....	273
Изменение данных с помощью табличных выражений .....	278

---

Изменение данных с использованием параметров TOP и OFFSET-FETCH .....	280
Инструкция OUTPUT .....	283
Сочетание с командой INSERT .....	283
Сочетание с командой DELETE .....	284
Сочетание с командой UPDATE .....	286
Сочетание с командой MERGE .....	287
Компоновка элементов языка DML .....	288
В заключение .....	290
Упражнения .....	290
Решения .....	293
<b>Глава 9. Транзакции и параллелизм .....</b>	<b>298</b>
Транзакции .....	298
Блокировки и блокирование .....	301
Блокировки .....	301
Проблемы, связанные с блокированием .....	304
Уровни изоляции .....	311
READ UNCOMMITTED .....	312
READ COMMITTED .....	313
REPEATABLE READ .....	314
SERIALIZABLE .....	316
Уровни изоляции, основанные на управлении версиями строк ..	317
Краткий обзор всех уровней изоляции .....	324
Взаимное блокирование .....	324
В заключение .....	327
Упражнения .....	327
<b>Глава 10. Программируемые объекты .....</b>	<b>339</b>
Переменные .....	339
Пакеты .....	341
Анализ .....	342
Пакеты и переменные .....	342
Команды, которые не могут находиться в одном пакете с другими командами .....	343
Разрешение имен .....	343
Параметр команды GO .....	344
Управление потоком выполнения .....	345
Инструкция IF...ELSE .....	345
Инструкция WHILE .....	346
Пример использования инструкций IF и WHILE .....	348
Курсоры .....	348
Временные таблицы .....	352
Локальные таблицы .....	352
Глобальные таблицы .....	354

---

Табличные переменные .....	356
Табличные типы.....	357
Динамические возможности языка SQL .....	358
Команда EXEC .....	359
Хранимая процедура sp_executesql.....	359
Оператор PIVOT .....	360
Процедуры.....	362
Пользовательские функции.....	362
Хранимые процедуры.....	364
Триггеры .....	366
Обработка ошибок.....	369
В заключение.....	373
<b>Приложение. Приступаем к работе .....</b>	<b>374</b>
Начинаем работу с SQL Database .....	374
Установка локального экземпляра SQL Server.....	375
1. Загрузка SQL Server .....	375
2. Создание учетной записи .....	375
3. Установка необходимых инструментов .....	376
4. Установка ядра базы данных, документации и утилит.....	376
Загрузка исходного кода и подготовка демонстрационной базы данных .....	384
Работа со средой SQL Server Management Studio.....	386
Работа с электронной документацией.....	391
<b>Алфавитный указатель .....</b>	<b>395</b>

# ПРЕДИСЛОВИЕ

Я очень счастлив, что Ицик нашел время и силы для написания книги об основах языка T-SQL. Работа преподавателем, руководителем и консультантом в области Microsoft SQL Server на протяжении многих лет помогает ему простым языком объяснять довольно сложные вопросы не только новичкам и неопытным пользователям, но и тем специалистам, чья работа с SQL Server не требует тесного знакомства с языком запросов.

Ицик считается одним из лучших мировых экспертов по языку T-SQL. Фактически, мы (члены команды разработки SQL Server) консультируемся с ним по поводу большинства новых синтаксических элементов, которые планируется ввести в ближайшем будущем. Его рекомендации и советы — важная часть процесса разработки SQL Server.

Для любого знатока своего дела написание ознакомительной литературы — это всегда непростая задача. Однако Ицику немало помогает богатый преподавательский опыт, ведь целевой аудиторией на его лекциях являются не только продвинутые, но и начинающие пользователи. Он смог выделить основы языка T-SQL и отойти от более углубленных тем. Однако это вовсе не означает, что из книги был выброшен абсолютно весь сложный материал. Ицик не боится доносить до читателя такие непростые темы, как теория множеств, исчисление предикатов и реляционная модель. Напротив, он пытается описать все это простыми словами, доходчиво объясняя их значимость для языка SQL. В результате читатель не только получает представление о механизме работы языка T-SQL, но и понимает, почему он был реализован именно так, а не иначе.

Не секрет, что лучшее средство подачи материала в книгах и справочниках по программированию — это удачный пример. В этом издании их будет немало, все необходимое вы сможете загрузить по ссылке [eksmo.ru/smv/TSQLFundamentals.rar](http://eksmo.ru/smv/TSQLFundamentals.rar). T-SQL — диалект языка SQL, стандартизированного организациями ISO и ANSI. Он содержит дополнения, которые помогают сделать программный код более выразительным и компактным. Многие примеры, которые приводит Ицик, имеют две версии: в одной используются фирменные средства языка T-SQL, а в другой предоставляется стандартное решение. Такой подход крайне удобен для читателей, которые уже знакомы со стандартом ANSI SQL, а также для программистов, стремящихся писать кроссплатформенный код.

О тесных связях Ицика с командой разработки SQL Server можно судить уже по первой главе, где описываются особенности разных вариантов поставки этого продукта — аппаратно-программного (Appliance), пакетного (Box) и облачного (Cloud). Ранее термин ABC (сокращенно от Appliance, Box и Cloud) не использовался за пределами компании Microsoft, но теперь нет никаких сомнений в том, что скоро он станет общеупотребительным. Приведенные примеры выверены на локальной и облачной версиях. Информацию о том, с чего следует начинать работу

с сервисом Windows Azure SQL Database, можно найти в приложении, расположенном в конце книги. Таким образом, это издание может стать для вас первым шагом на пути освоения облачных технологий, тем более что сайт Windows Azure предоставляет возможность бесплатной подписки.

Облачное расширение SQL Server чрезвычайно важно для изучения. Чтобы подчеркнуть его значение, придется отступить от общепринятых правил и прорекламировать в предисловии другую книгу — «Великий переход», автором которой является Николас Дж. Карр (издательство «Манн, Иванов и Фербер», 2013). В ней прорыв в облачных вычислениях сравнивается с процессом электрификации начала прошлого века. Моя убежденность в перспективности этих технологий еще больше окрепла после просмотра презентации Cloud Computing Economies of Scale («Облачные вычисления: экономия за счет масштаба»), которую Джеймс Гамильтон подготовил для конференции MIX10 (запись доступна по ссылке [channel9.msdn.com/events/MIX/MIX10/EX011](http://channel9.msdn.com/events/MIX/MIX10/EX011))<sup>1</sup>.

В своей книге Ицик упоминает об еще одном факторе, который напрямую связан с развитием облачных сервисов. Раньше версии SQL Server обновлялись один раз в несколько лет, но сейчас подобный подход уходит в прошлое. Будьте готовы к тому, что каждый год в вычислительных центрах компании Microsoft происходит несколько служебных обновлений. Различия между двумя версиями T-SQL (используемыми в SQL Server и Windows Azure SQL Database) предусмотрительно публикуются на сайте [tsql.solidq.com](http://tsql.solidq.com). Ицик сознательно не добавляет эту информацию в книгу, чтобы она не потеряла своей актуальности.

Приятного чтения. Надеюсь, вы в полной мере насладитесь знакомством с языком T-SQL.

---

<sup>1</sup> Все указанные в книге сайты англоязычные. Издательство не несет ответственности за их содержимое и напоминает, что со времени написания книги сайты могли измениться или вовсе исчезнуть. — *Примеч. ред.*

# ВВЕДЕНИЕ

В книге вы познакомитесь с языком запросов T-SQL (Transact-SQL), который используется в сервере баз данных Microsoft SQL Server и является разновидностью стандартов ISO и ANSI для языка SQL. Вы изучите теорию, лежащую в основе этой технологии, узнаете, как писать код для получения и изменения данных, а также получите общее представление о программируемых объектах.

Издание предназначено для начинающих, но вовсе не ограничивается набором пошаговых инструкций. По мере изучения синтаксиса запросов вы углубитесь в логику языка T-SQL и его элементов.

Некоторые темы будут сложны для тех, кто впервые столкнулся с технологией T-SQL. Такие разделы являются необязательными. Смело беритесь за них, если у вас не возникает трудностей с остальным материалом. В противном случае вы можете вернуться к ним попозже, когда накопите опыт. Факультативные разделы для углубленного чтения выделены особым образом.

Язык SQL содержит множество уникальных аспектов и сильно отличается от других языков программирования. Книга поможет понять суть данной технологии и ее составляющих. Вы научитесь оперировать множествами и усвоите стиль SQL-программирования.

Издание не привязано к конкретным версиям программного обеспечения, но в ней рассматриваются элементы языка, которые появились в последних модификациях SQL Server, в том числе в SQL Server 2012. При рассмотрении таких элементов будет указано, когда именно они введены.

База данных SQL Server доступна не только локально, но и в виде облачного сервиса под названием Windows Azure SQL Database (ранее SQL Azure). Примеры из книги тестировались на обоих вариантах сервера. О проблемах с совместимостью (например, некоторые возможности SQL Server 2012 пока недоступны в SQL Database) можно узнать на сайте компании SolidQ ([tsql.solidq.com](http://tsql.solidq.com)).

Усвоить информацию помогают приведенные упражнения. С их помощью вы сможете закрепить приобретенные знания на практике. Вы также встретите более сложные задачи, ориентированные на читателей, которые уверенно усваивают материал. Они помечены соответствующим образом и не обязательны для выполнения.

## Для кого предназначена книга

Издание рассчитано на разработчиков, использующих технологии T-SQL, DBAs и BI, специалистов по созданию отчетов, аналитиков, архитекторов и пользователей, которые только начинают работать с базой данных SQL Server и хотят научиться писать запросы и программировать на языке Transact-SQL.

Чтобы извлечь из книги максимум пользы, вы должны иметь опыт работы с операционной системой Windows и ее приложениями, а также владеть основными понятиями, которые относятся к системам управления реляционными базами данных.

## Кому книга противопоказана

Одна и та же книга не может быть универсальной и подходить для любой аудитории. В этом издании рассматриваются основы. Предлагаемый материал рассчитан в большей степени на людей, которые не знакомы с языком T-SQL или имеют незначительный опыт в его использовании. Тем не менее несколько читателей предыдущего издания отметили, что, несмотря на многолетнюю практику, они заполнили определенные пробелы в знаниях.

## Как устроена книга

Глава 1 — это фундамент, на который опирается весь остальной материал книги. Она содержит теоретические основы, здесь рассматриваются написание запросов и программирование на языке T-SQL, создание таблиц и обеспечение целостности данных. Различным аспектам получения и редактирования информации посвящены главы 2–8, в главе 9 затронута тема параллельных соединений и транзакций. В главе 10 вы познакомитесь с программируемыми объектами. Ниже приведено краткое содержание каждой части книги.

**Глава 1** «Общие сведения о создании запросов и программировании на языке T-SQL» посвящена основам языка SQL, теории множеств и исчислению предикатов. В ней исследуются такие темы, как реляционная модель, архитектура SQL Server, процесс создания таблиц и обеспечение целостности данных.

**Глава 2** «Запросы к одиночным таблицам» охватывает различные аспекты извлечения данных из одной таблицы с помощью оператора SELECT.

В **главе 3** «Соединения» рассматривается создание запросов к нескольким таблицам с помощью соединений (в том числе перекрестных, внутренних и внешних).

В **главе 4** «Вложенные запросы» речь пойдет о том, как заворачивать одни запросы в другие.

**Глава 5** «Табличные выражения» посвящена производным таблицам и обобщенным табличным выражениям, представлениям, вложенным функциям, возвращающим табличные значения, а также оператору APPLY.

**Глава 6** «Операторы работы с наборами» охватывает такие конструкции языка SQL, как UNION, INTERSECT и EXCEPT.

В **главе 7** «Продвинутые запросы» основное внимание уделяется оконным функциям, операторам PIVOT и UNPIVOT, работе со сгруппированными наборами данных.

В **главе 8** «Изменение данных» говорится о добавлении, обновлении, удалении и объединении содержимого таблиц.



В главе 9 «Транзакции и параллелизм» рассматриваются параллельные соединения, которые могут одновременно работать с одними и теми же данными. Вы познакомитесь с транзакциями, блокировками, уровнями изоляции и взаимным блокированием.

Глава 10 «Программируемые объекты» содержит общие сведения о программировании сервера SQL Server с помощью языка T-SQL.

В конце книги находится приложение «Приступаем к работе». В нем описано, как подготовить среду разработки, загрузить исходный код для представленных упражнений, установить демонстрационную базу данных TSQL2012 и написать запросы для SQL Server. Кроме того, в нем приводятся инструкции по работе с электронной документацией.

## Системные требования

В приложении указано, какие варианты SQL Server 2012 можно использовать для работы с кодом, представленным в издании. Они отличаются особыми требованиями к программному и аппаратному обеспечению. Более подробнее об этом вы прочтете в разделе «Требования к оборудованию и программному обеспечению для установки SQL Server 2012» электронной документации.

Для сервиса SQL Database эти требования не актуальны, поскольку обслуживанием его аппаратных и программных мощностей занимается компания Microsoft.

## Примеры кода

У книги есть свой сайт [eksmo.ru/smv/TSQLFundamentals.rar](http://eksmo.ru/smv/TSQLFundamentals.rar). Здесь вы найдете все примеры исходного кода, список ошибок и опечаток, а также дополнительные ресурсы. Подробнее об этом можно прочесть в приложении «Приступаем к работе».

## Благодарности

В создании книги прямо или опосредовано участвовало много людей, которые заслуживают отдельного упоминания.

Спасибо Лиле. Она наполняет смыслом все, что я делаю, и терпит меня, несмотря на бесчисленное количество времени, которое я уделяю языку SQL.

Спасибо за постоянную поддержку моим родителям, Миле и Гэби, а также брату Микки и сестре Инне. Вы смирились с моим отсутствием, хотя сейчас оно ощущается острее всего. Мама, мы все рассчитываем на твою доброту, силу и решимость. Папа, на тебя всегда можно положиться.

Выражаю благодарность членам команды разработки Microsoft SQL Server: Любору Коллару, Тобиасу Терншторму, Умачандару Джаячандрану и многим другим. Спасибо за ваши старания и время, которое вы потратили на встречи со мной, переписку по электронной почте, обдумывание ответов на мои вопросы и просьбы разъяснить некоторые моменты. Я считаю, что программные продукты SQL Server

2012 и SQL Database стали отличным подспорьем для дальнейшего развития языка T-SQL.

Выражаю признательность редакционным коллективам издательств O'Reilly Media и Microsoft Press, которые являются инициаторами этого проекта. Спасибо Расселу Джонсу, руководившему процессом со стороны издательства O'Reilly, а также Кристену Боргу, Кэти Краус и всем, кто принимал участие в работе над книгой.

Благодарю технических редакторов — Герберта Альберта и Джанлуку Хотц. Ваши правки, несомненно, улучшили качество и точность издания.

Спасибо компании SolidQ, в которой я работаю последние десять лет. Мне очень приятно быть ее частью и наблюдать за тем, как она развивается. Многие коллеги стали для меня партнерами, друзьями и членами семьи. Спасибо Фернандо Г. Герреро, Дугласу Макдауэллу, Герберту Альберту, Деян Шарка, Джанлуке Хотц, Жанне Ривз, Гленн Маккоин, Фрицу Лехницу, Эрику ван Солдту, Джоэл Бадд, Яну Тейлору, Мэрилин Темплтон, Берри Уокеру, Альберто Мартину, Лорене Хименес, Рону Талмейджу, Энди Келли, Рашабу Мехта, Эладио Ринкону, Эрику Веерману, Джей Хакни, Ричарду Уэймайру, Карлу Рабелеру, Крису Рэндаллу, Йохану Алену, Раулю Ийешу, Питеру Ларссону, Питеру Майерсу, Полу Терли и многим другим.

Я благодарен коллективу редакции журнала SQL Server Pro, особенно Меган Келлер, Лавон Питерс, Мишель Крокетт и Майку Отею. Больше десяти лет я был частью команды и мог делиться своими знаниями с читателями этого издания.

Отдельной благодарности заслуживают специалисты в области SQL Server — Алехандро Меса, Эрланд Держанский, Аарон Бертран, Тибор Карасци и Пол Вайт, а также Саймон Тьен, ведущий специалист. Мне очень приятно участвовать вместе с ними в программе MVP (Most Valuable Professional — наиболее ценный специалист). Уровень квалификации этих людей просто поражает, поэтому я всегда рад встретиться с ними, чтобы обменяться идеями или просто пообщаться за кружкой пива. Считаю, что нововведения в языке T-SQL в значительной мере определяются усилиями участников программы MVP и членами сообщества, объединенных интересом к данной технологии. Здорово, что совместная работа приводит к таким важным результатам.

И наконец, спасибо моим студентам за их прекрасные вопросы, которые заставляют меня совершенствоваться. Преподавание SQL дополнительно меня мотивирует. Это моя страсть и мое призвание.

# Глава 1

## ОБЩИЕ СВЕДЕНИЯ О СОЗДАНИИ ЗАПРОСОВ И ПРОГРАММИРОВАНИИ НА ЯЗЫКЕ T-SQL

Предлагаем вам погрузиться в мир, не похожий ни на один другой, — мир, в котором действуют особые законы. Если это ваше первое знакомство с языком **Transact-SQL (T-SQL)**, вы должны чувствовать себя, как Алиса перед началом приключений в Стране Чудес. У вас впереди все самое интересное! Для меня путешествие еще не закончилось. Напротив, я продолжаю свой путь, наполненный новыми открытиями.

Я работаю с языком T-SQL много лет: преподавание, презентации, написание статей и книг, консалтинг. Для меня это не просто язык, а способ мышления. Я часто имел дело со сложным материалом, но об основах не писал давно. Не потому, что это слишком легко, — на самом деле внешняя простота языка T-SQL обманчива. Я мог бы поверхностно объяснить его синтаксис, чтобы вы уже через несколько минут создавали собственные запросы, но в долгосрочной перспективе такой подход только повредит, мешая понять суть технологии.

Выступая проводником в новом для вас мире, я беру на себя большую ответственность. Поэтому, прежде чем писать об основах, хочется быть уверенным, что потрачено достаточно времени и сил на изучение и понимание данной темы. T-SQL — глубокий язык, и знакомство с ним следует начинать не с базового синтаксиса и написания запросов, которые возвращают нужный результат. Фактически, вы должны забыть все, что знали о программировании, и начать мыслить понятиями, принятыми в T-SQL.

### Теоретические основы

SQL расшифровывается как Structured Query Language (структурированный язык запросов). Это стандартный язык, специально созданный для выполнения запросов и управления содержимым реляционных баз данных (БД). Система управления реляционными базами данных (СУРБД) основывается на реляционной (логической) модели (РМД) представления информации, которая в свою очередь базируется на двух разделах математики: теории множеств и исчислении предикатов. Многие языки программирования и целые области информатики развивались интуитивно, но технология SQL имеет под собой прочный фундамент — прикладную математику. Таким образом, язык T-SQL опирается на крепкую теоретическую базу. Компания Microsoft позиционирует его как диалект (или расширение) языка SQL для своей фирменной СУРБД под названием Microsoft SQL Server.

В этом разделе мы познакомимся с теоретическими основами языка SQL, обсудим теорию множеств, исчисление предикатов, реляционную модель и жизненный цикл информации. Книга не является учебником по математике или проектированию/моделированию БД, поэтому представленные сведения лишены многих формальностей и не претендуют на полноту. Моя задача — дать базу для изучения языка T-SQL и очертить ключевые моменты, необходимые для правильного понимания дальнейшего материала.



### НЕЗАВИСИМОСТЬ ОТ ЯЗЫКА

Реляционная модель не привязана к конкретному языку. Вы можете реализовать ее в любом языке программирования, например в рамках структуры классов C#. Сейчас многие СУРБД помимо диалектов SQL поддерживают другие средства управления (к примеру, в SQL Server предусмотрена интеграция со средой CLR).

Кроме того, следует понимать, что SQL в некоторых аспектах отклоняется от реляционной модели. Есть мнение, что ему на смену должен прийти новый язык, который будет следовать постулатам РМД более строго. Однако на данный момент SQL является промышленным стандартом и применяется во всех ведущих СУРБД.



### ДОПОЛНИТЕЛЬНЫЙ МАТЕРИАЛ

Подробности о расхождениях языка SQL и реляционной модели данных, а также сведения об использовании SQL в рамках РМД можно найти во втором издании книги Кристофера Дейта SQL and Relational Theory: How to Write Accurate SQL Code (издательство O'Reilly Media, 2011).

## SQL

В начале 1970-х гг. компания IBM разработала язык программирования SEQUEL (Structured English QUery Language — структурированный английский язык запросов), который использовался в ее фирменной СУРБД под названием System R. Позже из-за споров, связанных с торговой маркой, имя языка было изменено на SQL. В 1986 г. SQL стал стандартом Американского национального института стандартов (ANSI), а через год прошел стандартизацию Международной организации по стандартизации (ISO). С тех пор раз в несколько лет ANSI и ISO выпускают новые версии языка. За это время свет увидели следующие стандарты: SQL-86 (1986), SQL-89 (1989), SQL-92 (1992), SQL:1999 (1999), SQL:2003 (2003), SQL:2006 (2006), SQL:2008 (2008) и SQL:2011 (2011).

SQL напоминает английский язык и имеет логическую структуру. В отличие от других языков программирования, которые используют императивный подход, в SQL применяется декларативный стиль. Другими словами, с помощью SQL описывается, *что и как* вы хотите получить, а все действия по обработке запроса выполняет СУРБД.

SQL сочетает элементы языков **DDL** (Data Definition Language — язык описания данных), **DML** (Data Manipulation Language — язык управления данными) и **DCL** (Data Control Language — язык управления БД). DDL описывает объекты и содержит инструкции CREATE, ALTER, DROP и т. д. Язык DML, который

позволяет запрашивать и редактировать данные, включает операторы `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` и `MERGE`. Бытует заблуждение, что DML ограничивается лишь командами для редактирования, но, как упомянуто выше, в состав языка входит инструкция `SELECT`. Кроме того, многие считают, что оператор `TRUNCATE` относится к DDL, хотя он является частью DML. Язык DCL предназначен для работы с правами доступа и содержит инструкции `GRANT` и `REVOKE`. В книге мы сосредоточимся на языке DML.

T-SQL — диалект языка SQL, который предоставляет некоторые фирменные расширения. При знакомстве с каждым элементом синтаксиса я буду отдельно отмечать, стандартный он или нет.

## Теория множеств

Теория множеств, разработанная Георгом Кантором, — один из разделов математики, на которых основана реляционная модель. Вот как Кантор определяет множество:

*Под множеством мы понимаем любое объединение в одно целое определенных, вполне различных объектов нашей интуиции или нашей мысли.*

— Джозеф У. Даубен и Георг Кантор  
(издательство Принстонского университета, 1990)

Каждое слово в этой цитате имеет глубокий смысл. Определения множества и принадлежности к нему являются аксиомами. Любой элемент вселенной либо входит в множество, либо нет.

Начнем со словосочетания «в одно целое». Множество следует рассматривать как единую сущность. Вы должны сосредоточиться на совокупности объектов, а не на отдельных ее составляющих. Позже, когда вы будете писать запросы к таблице (например, таблице сотрудников), вам придется думать о ее содержимом как о едином целом, игнорируя частные записи. Это звучит довольно просто, однако, как показывает практика, многим программистам такой образ мышления дается тяжело.

Слово «различимых» означает, что каждый элемент множества должен быть уникальным. Обеспечить уникальность строк в таблице можно с помощью ключа. Таблицу, у которой его нет, нельзя считать множеством, поскольку строки невозможно идентифицировать уникальным образом (это уже будет **мультимножество**).

Выражение «нашей интуиции или нашей мысли» подразумевает, что определение множества является субъективным. Возьмем классную комнату: кто-то представляет ее как множество людей, а кому-то она кажется сочетанием двух множеств — учеников и учителей. Таким образом, в трактовке этого понятия есть определенная свобода. Проектируя модель данных для БД, необходимо тщательно взвешивать субъективные потребности приложения, чтобы подобрать подходящее определение будущих записей.

Что касается слова «объектов», то определение множества не ограничивается физическими сущностями, такими как автомобили или сотрудники; оно также оперирует абстрактными понятиями вроде простых чисел или линий.

Однако то, что не учел Кантор, является не менее важным, чем его определение. Обратите внимание, в цитате ничего не сказано о порядке размещения элементов, поскольку с точки зрения теории множеств он не играет никакой роли. Для перечисления элементов множества используют фигурные скобки: {a, b, c}. Поскольку порядок не имеет значения, одно и то же множество можно записать по-разному: {b, a, c} или {b, c, a}. Говоря о множестве атрибутов (в SQL их называют **столбцами**), которые составляют заголовок отношения (в SQL это **таблица**), отмечу, что элемент должен идентифицироваться по имени, а не по порядковому номеру.

Таким же образом рассмотрим множество кортежей (в SQL — **строки**), составляющее тело отношения. Здесь каждый элемент идентифицируется значением своего ключа, а не позицией. Многие программисты с трудом принимают тот факт, что строки в таблице, к которой они выполняют запрос, никак не упорядочены. Иными словами, если вы специально не укажете определенный способ сортировки данных, записи с результатами запроса будут размещены в произвольном порядке.

## Исчисление предикатов

Еще один раздел математики — исчисление предикатов — изучается со времен Древней Греции. Основоположник реляционной модели Эдгар Ф. Кодд связал исчисление предикатов с управлением данными и процессом их извлечения. Проще говоря, **предикат** — это свойство или выражение, которое либо имеет место, либо нет (то есть оно либо истинное, либо ложное). С помощью предикатов в реляционной модели поддерживается логическая целостность и структурированность данных. Представьте, что у нас есть таблица под названием **Employees**, которая хранит записи только о служащих с зарплатой больше нуля. Ее предикатом будет «зарплата больше нуля» (выражение на языке T-SQL выглядит как `зарплата > 0`).

Вы можете использовать предикаты для различных задач, например, чтобы определить подмножества, фильтруя данные. Если в результате запроса к таблице **Employees** необходимо получить строки только для сотрудников из отдела продаж, предикатом для вашего фильтра выступит фраза «отдел равен продажам» (выражение на языке T-SQL: `отдел = 'продажи'`).

Предикаты используют и для определения множества, ведь его не всегда можно найти путем перечисления всех элементов (например, если оно бесконечно). Данный подход также позволяет квалифицировать множества на основе их свойств. Примером бесконечного множества, определенного с помощью предиката, служит последовательность простых чисел; предикат в этом случае звучит так: «x — положительное целое число больше 1, которое делится только на 1 и на само себя». Для любого указанного значения предикат будет либо истинным, либо ложным. Множество простых чисел — это набор всех элементов, для которых предикат является истинным. Этим способом находят и конечные множества. Например, для набора элементов {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} истинный предикат выглядит следующим образом: «x — целое число, которое больше или равно 0 и меньше или равно 9».

## Реляционная модель

Реляционная (логическая) модель, предназначенная для управления и манипулирования данными, основывается на теории множеств и исчислении предикатов. Она была создана Эдгаром Ф. Коддом, а позже истолкована и усовершенствована

Кристофером Дейтом, Хью Дарвенем и др. Первую версию реляционной модели Кодд предложил в 1969 г. в исследовательском отчете компании IBM, который назывался «Дифференцируемость, избыточность и согласованность отношений, хранящихся в больших хранилищах данных». В 1970 г. журнал Communications of the ACM опубликовал научную работу под заголовком «Реляционная модель данных для больших разделяемых хранилищ», где Кодд изложил доработанную версию своей идеи.

Задача реляционной модели — обеспечить согласованное представление данных с минимальной (или вовсе отсутствующей) избыточностью, не жертвуя при этом их полнотой; целостность данных, вытекающая из их согласованности, является частью модели. Помимо реализации реляционной модели СУРБД должна предоставлять средства для хранения данных, управления ими, обеспечения их целостности и выполнения запросов к ним. Прочный математический фундамент позволяет с уверенностью сказать, имеет ли определенный экземпляр модели данных (на основе которой позже будет сгенерирована реальная БД) какие-либо изъяны.

Реляционная модель опирается на такие понятия, как предложения, предикаты, отношения, кортежи, атрибуты и т. д. Людям, далеким от математики, эти термины могут показаться устрашающими. В следующих разделах мы рассмотрим их ключевые аспекты, используя неформальный подход без сухих математических определений. Я также попытаюсь объяснить, как все это относится к БД.

## Предложения, предикаты и отношения

Мнение о том, что термин «реляционный» (relational) относится к связям между таблицами, не совсем корректно. На самом деле это математическое понятие, которое используется для представления множества. С точки зрения реляционной модели отношение — это множество связанной между собой информации; в языке SQL аналогом термина (хоть и не стопроцентным) является таблица. Ключевая особенность реляционной модели заключается в том, что одно отношение должно быть представлено одним множеством (например, таблицей Customers). Стоит отметить, что результатом операций с отношениями (основанными на реляционной алгебре) также будет отношение (например, при соединении двух таблиц получится новая таблица).



### ПРИМЕЧАНИЕ

В рамках реляционной модели различают понятия отношений и реляционных переменных, но я в обоих случаях буду использовать термин **отношение**. Кроме того, отношения состоят из заголовка и тела. Заголовок содержит множество атрибутов (столбцов), где каждый элемент идентифицируется именем и названием типа. Тело — это множество кортежей (строк), где любой элемент распознается с помощью ключа. Чтобы вас не путать, я буду говорить о таблице как о наборе строк.

Проектируя структуру БД, вы должны подать информацию в виде отношений (таблиц). Сначала необходимо определить предложения, которые будут представлены в вашей БД. Предложение — это истинное или ложное утверждение. Например, утверждение «служащий Ицик Бен-Ган родился 12 февраля 1971 г. и работает в отделе информационных технологий» является предложением. Истинное предложение подтверждается и принимает вид строки в таблице. Ложное предложение не может себя подтвердить. Такой подход называют **предположением о замкнутости мира** (closed world assumption, или CWA).

Далее предложения подлежат формализации. Для этого возьмите реальные данные (тело отношения) и определите их структуру (заголовок отношения), например формируя из предложений предикаты. Считайте, что предикаты — это параметризованные предложения. Заголовок отношения содержит набор атрибутов. Обратите внимание на применение слова «набор»: в реляционной модели атрибуты являются неупорядоченными и различимыми. Каждый атрибут идентифицируется по имени и названию типа. Например, заголовок отношения **Employees** может состоять из следующих атрибутов (выраженных в виде пар имя/название типа): `employeeid integer, firstname character string, lastname character string, birthdate date, departmentid integer`.

**Тип** — одна из наиболее фундаментальных составляющих отношений. Он ограничивает атрибут определенным набором возможных или корректных значений. Например, тип `INT` — множество всех целых чисел в диапазоне от 2 147 483 648 до 2 147 483 647. Это один из простейших видов предикатов в БД. К примеру, БД не примет предложение, в котором дата рождения сотрудника выглядит как «31 декабря 1971 г.» (не говоря о значении вроде «абвгд!»). Тип подходит не только для описания целых чисел или строк, он может перечислять возможные значения (скажем, должности). Тип также бывает сложным. Наверное, лучше всего о нем думать как о классе, состоящем из инкапсулированных данных и механизма, который этими данными управляет. В качестве примера можно привести геометрический тип, поддерживающий многоугольники.

## Отсутствующие значения

Источником многочисленных споров является утверждение, что предикаты в реляционной модели ограничены рамками двоичной логики. Другими словами, предикат может быть либо истинным, либо ложным — других состояний у него нет. Двоичная логика согласуется с законом исключенного третьего. Однако есть мнение, что предикат может поддерживать три (или даже четыре) состояния, чтобы учитывать случаи, когда значения отсутствуют. Такой предикат будет одновременно и не истинным, и не ложным — неизвестным.

Рассмотрим атрибут «мобильный телефон» в таблице **Employees**. Предположим, что у некоторых служащих нет мобильного номера. Как представить данный факт в БД? Троичная логика позволяет ставить для отсутствующего значения специальную отметку. И если телефон сотрудника не указан, предикат вернет «неизвестно». Исчисление предикатов в троичной системе предусматривает три возможных значения: истина, ложь и неизвестно.

Одни считают, что троичная логика не является реляционной, другие придерживаются противоположного мнения. Кодд выступал за четвертичную логику, аргументируя это тем, что существует два вида отсутствующих значений — применимые (A-Mark) и неприменимые (I-Mark). Примером отсутствующего, но применимого значения служит ситуация, когда у служащего есть мобильный телефон, но вы не знаете его номер. Неприменимое отсутствующее значение — когда у работника вовсе нет мобильного телефона. По мнению Кодда, в таких случаях необходимо использовать две специальные отметки. В языке SQL предикаты исчисляются в троичной системе, где третьим (неизвестным) значением выступает отметка `NULL`. Это вызывает путаницу и повышает сложность кода. Однако использование двоичной логики не менее проблематично.



## Ограничения

В реляционной модели есть одна замечательная особенность: она изначально поддерживает целостность данных. Это достигается за счет правил (**ограничений**), которые определяются в рамках модели данных и соблюдаются на уровне СУРБД. Простейший способ обеспечить целостность — назначить тип атрибута, указав сопутствующий параметр, который поддерживает отметку NULL. Ограничения также обеспечиваются на уровне самой модели. Например, отношение `Orders (orderid, orderdate, duedate, shipdate)` поддерживает хранение трех разных дат, а таблицы `Employees (empid)` и `EmployeeChildren (empid, childname)` позволяют устанавливать для каждого экземпляра `Employees` любое количество записей `EmployeeChildren` (от нуля до бесконечности).

Другим примером ограничений служат потенциальные и внешние ключи. Они обеспечивают соответственно сущностную и ссылочную целостность данных. **Потенциальный ключ** определяют для одного или нескольких атрибутов, чтобы исключить многократное вхождение в отношение одного и того же кортежа (строки). Предикаты, основанные на таком ключе, способны однозначно идентифицировать строку (запись о конкретном сотруднике). В одном отношении бывает несколько потенциальных ключей. Допустим, таблица `Employees` может содержать ключи для столбцов `employeeid`, `SSN` (номер социального страхования) и т. д. Обычно один из потенциальных ключей становится **первичным** (к примеру, столбец `employeeid` в таблице `Employees`) и затем применяется как предпочтительный способ идентификации строк. Тогда остальные потенциальные ключи называются **резервными**.

**Внешний ключ** определяется для одного или нескольких атрибутов и указывает на потенциальный ключ в другом или том же отношении (в таком случае первое отношение называют **ссылочным**). Это позволяет ограничивать атрибуты внешнего ключа в ссылочном отношении значениями, которые содержатся в соответствующих атрибутах потенциального ключа. Предположим, в таблице `Employees` для столбца `departmentid` определен внешний ключ, который ссылается на первичный ключ таблицы `Departments`, основанный на столбце `departmentid`. Значит, содержимое `Employees.departmentid` ограничено значениями столбца `Departments.departmentid`.

## Нормализация

Реляционная модель также определяет **правила нормализации** (их еще называют **нормальными формами**). Нормализация — формальный математический процесс, в результате которого каждая сущность обязательно должна быть представлена только одним отношением. Нормализованная БД не подвержена аномалиям, возникающим при редактировании содержимого; она обеспечивает полноту данных, сохраняя минимальную избыточность. Если вы работаете в рамках модели ERM (Entity Relationship Modeling, то есть сущность — связь), представляя каждое отношение вместе со всеми атрибутами, вам не нужно думать о нормализации; она может понадобиться лишь для усиления модели и обеспечения ее корректности. Рассмотрим первые три нормальные формы (1NF, 2NF и 3NF), введенные Эдгаром Коддом.

### 1NF

Согласно первой нормальной форме кортежи (строки) внутри отношения (таблицы) должны быть уникальными, а атрибуты атомарными. Другими словами, таблица, которая полностью соответствует отношению, имеет первую нормальную форму.

Исключительность строк достигается за счет определения в таблице уникального ключа.

Вы можете оперировать атрибутами только с помощью действий, которые определены как часть типа атрибута. Их атомарность субъективна в том же смысле, что и определение множества. К примеру, сколькими столбцами должно выражаться имя сотрудника в отношении **Employees**: одним (полное имя), двумя (имя и фамилия) или тремя (имя, фамилия и отчество)? Это зависит от ситуации. Если вы будете работать с отдельными составляющими имени (скажем, для выполнения поиска), необходимо разбить его на части; в противном случае делать этого не стоит.

Атрибут может обладать не только недостаточной, но и неполной атомарностью. Например, если мы считаем, что атрибут, содержащий адрес, является атомарным, то отсутствие названия города в нем будет нарушать первую нормальную форму.

Часто суть первой формы понимается неверно. Некоторые думают, что она нарушается при попытках имитировать массивы. Примером может быть отношение **YearlySales** с атрибутами **salesperson**, **qty2010**, **qty2011** и **qty2012**. Оно никак не нарушает первую нормальную форму; здесь просто накладывается ограничение, согласно которому данные распределяются по трем годам — 2010, 2011 и 2012.

## 2NF

Вторая нормальная форма вводит два правила: одно предусматривает обязательное соответствие первой форме, другое касается связи между атрибутами с потенциальным ключом и без него. Атрибуты, для которых не определено ни одного ключа, должны полностью функционально зависеть от атрибутов с потенциальными ключами. Если вы хотите получить содержимое атрибута без ключа, вам необходимо предоставить значения всех атрибутов кортежа, на основе которого сформирован потенциальный ключ. Зная данные столбцов потенциального ключа, вы получите содержимое любого столбца в любой строке.

В качестве примера нарушения второй нормальной формы представим ситуацию: вы определили отношение под названием **Orders**, которое представляет заказы и их составляющие (рис. 1.1). Таблица **Orders** содержит следующие столбцы: **orderid**, **productid**, **orderdate**, **qty**, **customerid** и **companyname**. Первичный ключ определен для атрибутов **orderid** и **productid**.

Orders	
PK	<u>orderid</u>
PK	<u>productid</u>
	orderdate
	qty
	customerid
	companyname

**Рис. 1.1.** Модель данных до применения второй нормальной формы

Как видно на рисунке 1.1, атрибуты, для которых не определен ключ, зависят только от части потенциального ключа (в нашем случае он является также первичным), что нарушает вторую нормальную форму. Например, с помощью одного лишь идентификатора **orderid** вы найдете значения атрибутов **orderdate**,

`customerid` и `companyname`. Чтобы нормализовать таблицу, следует разделить ее на две части: **Orders** и **OrderDetails** (рис. 1.2). Первое отношение, **Orders**, будет включать атрибуты `orderid` (с первичным ключом), `orderdate`, `customerid` и `companyname`. Атрибуты `orderid`, `productid` и `qty` отойдут отношению **OrderDetails**; первые два из них сформируют первичный ключ.

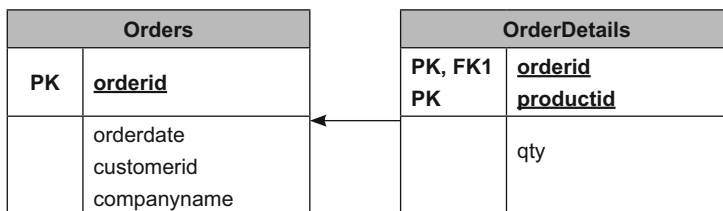


Рис. 1.2. Модель данных после применения 2NF, но без учета 3NF

### 3NF

Третья нормальная форма тоже имеет два правила. Первое — данные должны соответствовать второй нормальной форме. Второе — зависимость всех атрибутов, для которых не определен ключ, от потенциального ключа должна быть не транзитивной. Проще говоря, атрибуты без ключа должны быть независимыми друг от друга.

Теперь отношения **Orders** и **OrderDetails**, описанные выше, соответствуют второй нормальной форме. Напомним, что на этом этапе таблица **Orders** состоит из столбцов `orderid`, `orderdate`, `customerid`, `companyname` и первичного ключа на основе идентификатора `orderid`, от которого зависят атрибуты `customerid` и `companyname`. Чтобы получить содержимое атрибута `customerid`, который представляет клиента, сделавшего заказ, вам необходимо предоставить весь первичный ключ. То же относится к столбцу `companyname`, связанному с клиентом. Однако атрибуты `customerid` и `companyname` зависят друг от друга. Обеспечить соответствие третьей нормальной форме можно, если ввести новое отношение под названием **Customers** (рис. 1.3), которое будет содержать атрибуты `customerid` (в качестве первичного ключа) и `companyname`. После этого убираем столбец `companyname` из таблицы **Orders**.

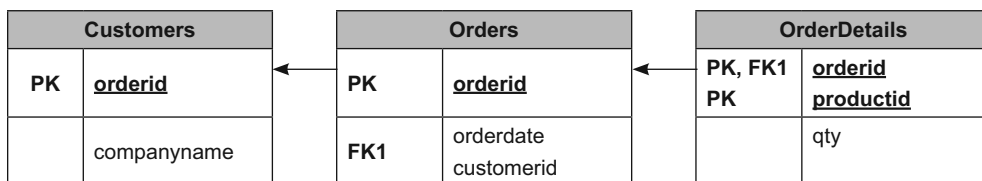


Рис. 1.3. Модель данных после применения третьей нормальной формы

Если отбросить формальности, суть второй и третьей форм можно сформулировать следующим образом: «Каждый атрибут, для которого не определен ключ, зависит от целого ключа, только от ключа и ни от чего, кроме ключа».

Помимо трех классических нормальных форм, определенных Эдгаром Коддом, есть и другие, более высокоуровневые. В них речь идет о составных ключах и темпоральных (временных) БД, что уже выходит за рамки нашей книги.

## Жизненный цикл данных

Под данными обычно понимают нечто статическое — их добавляют в БД, затем они возвращаются в виде запросов. Во многих системах данные похожи на продукт, который собирается на конвейере, двигаясь от одного механизма к другому; при этом на каждом этапе жизненного цикла данные могут менять свои характеристики и влиять на окружающую их среду. Жизненный цикл данных представлен на рисунке 1.4.

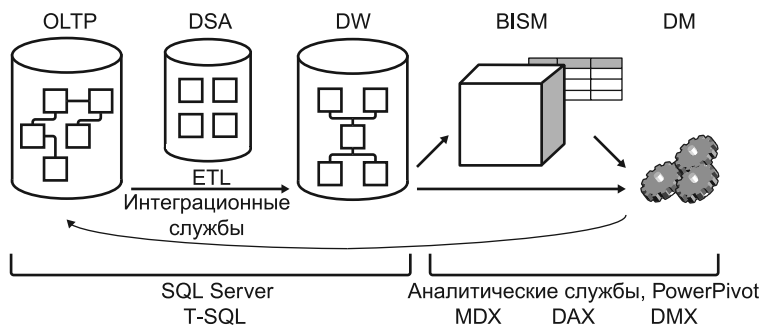


Рис. 1.4. Жизненный цикл данных

Приведем краткое описание сокращений:

- OLTP (Online Transactional Processing) — обработка транзакций в реальном времени;
- DSA (Data Staging Area) — промежуточная область;
- DW (Data Warehouse) — хранилище данных;
- BISM (Business Intelligence Semantic Model) — семантическая модель бизнес-аналитики;
- DM (Data Mining) — интеллектуальный анализ данных;
- ETL (Extract, Transform, Load) — извлечение, преобразование, загрузка;
- MDX (Multidimensional Expressions) — многомерные выражения;
- DAX (Data Analysis Expressions) — выражения для анализа данных;
- DMX (Data Mining Extensions) — расширения интеллектуального анализа данных.

## Обработка транзакций в реальном времени

Изначально данные поступают в систему обработки транзакций (OLTP). Этот механизм предназначен для ввода данных: транзакции выполняют в основном операции добавления, обновления и удаления. Реляционная модель рассчитана в первую очередь на системы, где нормализованные отношения обладают хорошей производительностью и согласованностью. В таких условиях каждая таблица представляет отдельную сущность, сводя избыточность к минимуму. Изменение записи всегда происходит только в одном месте, что снижает вероятность возникновения ошибок и обеспечивает высокую скорость работы при редактировании данных.

Однако OLTP не подходит для операций извлечения, поскольку нормализованная модель обычно состоит из множества таблиц (по одной на каждую сущность) со сложными взаимосвязями. Даже для создания простого отчета потребуются выполнить большое количество операций соединения, а в результате получится громоздкий и медленный запрос.

Вы можете реализовать БД типа OLTP с помощью SQL Server; управление ее содержимым и создание запросов к ней будут выполняться с помощью языка T-SQL.

## Хранилище данных

**Хранилище данных** — это система, которая предназначена для извлечения и вывода информации. Она целиком обслуживает какую-либо административную единицу (например, компанию или офис). Хранилище части организации (отдела) или определенной тематики называется **витриной данных**. Модель хранилища данных оптимизирована для операций извлечения, поэтому обладает повышенной избыточностью, упрощенными связями и содержит меньшее количество таблиц. Как результат — более простые и эффективные запросы по сравнению с системой OLTP.

В простейшем случае хранилище данных имеет так называемую **структуру звезды** — состоит из нескольких таблиц измерений и одной таблицы фактов. Каждая таблица измерений представляет тематику, в рамках которой должен проводиться анализ информации. Например, если у вас есть система управления заказами и продажами, вы, вероятно, захотите анализировать данные с точки зрения клиентов, продуктов, сотрудников, времени и т. д. При использовании звездоподобной структуры каждое измерение имеет отдельную таблицу с избыточным содержимым. Тогда измерение «продукт» можно реализовать в виде единого отношения, не разбивая его на три нормализованные таблицы (для категорий, подкатегорий и самих продуктов). Если применять нормализацию, вы получите **структуру снежинки** вместо структуры звезды.

Таблица фактов содержит определенные сведения или числовую характеристику (для каждой подходящей комбинации ключей в таблицах измерений). К примеру, каждая комбинация покупатель — продукт — сотрудник — время должна иметь в таблице фактов одну запись со сведениями о количестве и цене. Данные в хранилище обычно подготавливаются заранее с определенной периодичностью (скажем, за сутки); в системах OLTP это, как правило, происходит во время транзакции.

Ранние версии SQL Server в основном ориентировались на системы OLTP, но в какой-то момент фокус сместился в сторону хранения и анализа данных. Вы можете организовать собственное хранилище на основе БД в SQL Server, используя для создания запросов и администрирования язык T-SQL.

Процесс, во время которого данные извлекаются из исходной системы (OLTP или другой), изменяются и загружаются в хранилище, называют ETL. Для управления им в SQL Server предусмотрена служба SSIS (SQL Server Integration Services).

Часто при выполнении ETL приходится использовать промежуточную область (DSA) между системой OLTP и хранилищем данных. Обычно DSA применяется в реляционных БД, выступая средством фильтрации. Это внутренний механизм, недоступный конечному пользователю.

## Семантическая модель бизнес-аналитики

BISM — новейшая модель от компании Microsoft, которая предназначена для полноценной поддержки приложений, занимающихся бизнес-аналитикой. Ее задача заключается в том, чтобы предоставить гибкий, эффективный и масштабируемый механизм для анализа и создания отчетов. Архитектура этой модели состоит из трех уровней:

- модели данных;
- бизнес-логики и запросов;
- доступа к данным.

Развертывание модели выполняется на сервере служб анализа (Microsoft Analysis Services) или с помощью системы PowerPivot. Службы анализа ориентированы на профессионалов в области бизнес-аналитики и информационных технологий; система PowerPivot больше подходит для промышленного потребления. Сервер Analysis Services позволяет использовать многомерные или табличные (реляционные) модели данных. PowerPivot поддерживает исключительно реляционную модель.

В бизнес-логике и запросах применяют два языка: MDX, основанный на принципах многомерности, и DAX, который оперирует таблицами.

На уровне доступа к данным могут использоваться разные источники: реляционные БД (например, хранилище данных), файлы, облачные сервисы, отраслевые приложения, потоки в формате OData и т. д. На этом уровне информация либо кэшируется, либо передается напрямую из источников. Кэшированная модель использует одну из двух систем хранения. Первая, MOLAP (Multidimensional Online Analytical Processing), поддерживает многомерную модель и выполняет предварительную загрузку данных. Вторая, VertiPaq, реализует индексы columnstore и обеспечивает крайне высокую степень сжатия. Это очень быстрая система, для которой не нужно выполнять предварительную загрузку данных, создавать отдельные индексы и т. д.



### ДОПОЛНИТЕЛЬНЫЙ МАТЕРИАЛ

Этот раздел содержит слишком много понятий для книги, посвященной основам языка T-SQL. Если вас заинтересовала концепция модели BISM, подробности о ней вы найдете в статье, авторами которой являются разработчики Microsoft Analysis Services ([blogs.msdn.com/b/analysisservices/archive/2011/05/16/analysis-services-vision-and-roadmap-update.aspx](http://blogs.msdn.com/b/analysisservices/archive/2011/05/16/analysis-services-vision-and-roadmap-update.aspx)).

## Интеллектуальный анализ данных

Модель BISM способна ответить на любой правильно поставленный вопрос, извлекая из множества данных аномалии, тенденции и прочую информацию. В процессе динамического анализа пользователь переходит от одной аналитической выборки к другой, пытаясь выделить лишь необходимые ему данные.

Однако существует более высокоуровневый подход: вместо того чтобы самостоятельно искать полезные сведения, можно прибегнуть к интеллектуальному анализу. Специальные алгоритмы исследуют данные и отбирают то, что нужно. Ин-

теллектуальный анализ представляет огромную ценность для бизнеса. Он помогает определять тенденции, выясняет, какие продукты часто приобретают вместе, предсказывает выбор покупателя на основе специфических параметров.

Система Analysis Services поддерживает алгоритмы интеллектуального анализа, включая кластеризацию, дерево принятия решений и т. д. Чтобы управлять моделями и извлекать данные в рамках этого процесса, используют язык DMX.

## Архитектура SQL Server

В этом разделе вы познакомитесь с архитектурой SQL Server, узнаете о разновидностях данного продукта и его составляющих — экземплярах сервера, БД, схемах, объектах БД, — а также о назначении каждой из них.

### Разновидности SQL Server

На протяжении многих лет у SQL Server была всего одна версия — коробочная (или локальная). Однако в последнее время компания Microsoft решила отойти от этой практики и позволила клиентам выбирать наиболее подходящую разновидность своего продукта. Сейчас SQL Server поставляется в трех вариантах: программно-аппаратный комплекс, коробочная версия и облачный сервис.

#### Программно-аппаратный комплекс

Идея этой версии состоит в том, чтобы предоставить полноценное решение, которое физически находится у клиента и помимо набора сервисов включает программное и аппаратное обеспечение. Сегодня существует несколько версий таких решений. Среди них можно выделить продукт под названием Parallel Data Warehouse (PDW). Чтобы добиться высоких показателей производительности, безопасности и доступности, специалисты Microsoft сотрудничают с корпорациями Dell и HP, которые изготавливают оборудование.

Вам, наверное, интересно, с помощью какого языка осуществляется взаимодействие с ядром БД? Это зависит от конкретного продукта. Например, в коробочных версиях SQL Server и PDW используются разные ядра. В специализированном ядре продукта PDW применяется особый диалект языка SQL под названием DSQL (distributed SQL — **распределенный SQL**). Компания Microsoft пытается сделать поддержку языков в своих продуктах более однородной, но пока цель не достигнута. В книге мы сосредоточим внимание на технологии T-SQL, которая поддерживается в коробочных, облачных и некоторых программно-аппаратных вариантах SQL Server.

#### Коробочная версия

Коробочная версия SQL Server, которую официально называют **локальной**, является традиционной и обычно устанавливается у клиента. Пользователь отвечает за подготовку оборудования, развертывание сервера, выполнение обновлений, а также за обеспечение высокой доступности, аварийного восстановления (система HADR), безопасности и всего остального.

Клиент может установить несколько экземпляров продукта на один компьютер (об этом поговорим в следующем разделе), выполняя запросы, которые взаимодействуют с несколькими БД. Есть возможность переключать соединение между разными БД (если они не автономны).

В качестве языка запросов в коробочной версии используется T-SQL. При желании вы можете запустить на своем локальном экземпляре SQL Server все примеры кода и упражнения, которые здесь приводятся. В дополнении к книге содержатся подробности о загрузке и установке пробной версии SQL Server; там вы найдете также инструкции по созданию демонстрационной БД.

## Облачные сервисы

Компания Microsoft предлагает две облачные разновидности SQL Server: частную и публичную. В первом случае сервис размещен локально, однако используется технология виртуализации. Здесь применяется то же ядро, что и в коробочной версии (то есть запросы пишутся на языке T-SQL), но с некоторыми ограничениями, которые накладывает виртуальная среда (это касается, например, количества процессоров и оперативной памяти).

Публичный облачный сервис называется Windows Azure SQL Database (бывший SQL Azure) и работает внутри удаленных центров обработки данных. За оборудование, обслуживание, аварийное восстановление и выполнение обновлений отвечает компания Microsoft. Клиенту необходимо лишь оптимизировать индексы и запросы.



### ПРИМЕЧАНИЕ

В дальнейшем вместо Windows Azure SQL Database мы будем использовать сокращенное название — SQL Database.

Продукт SQL Database позволяет создавать на облачном сервере несколько БД, однако подключаться к ним можно исключительно по очереди. В SQL Database используется особенное ядро, хотя в его основе лежит код из локальной версии, поэтому с точки зрения поддержки языка T-SQL эти разновидности SQL Server практически ничем не отличаются. Большинство возможностей T-SQL, которые вы изучите в книге, можно применять к облачному и коробочному вариантам, хотя существуют исключения.

Например, некоторые функции языка T-SQL из локальной версии не реализованы (или недоступны) в SQL Database. Подробнее это изложено в электронном справочнике на странице [msdn.microsoft.com/ru-ru/library/windowsazure/ee336281.aspx](https://msdn.microsoft.com/ru-ru/library/windowsazure/ee336281.aspx). Стоит отметить, что в облачном сервисе программное обеспечение обновляется и разворачивается чаще, чем в коробочном варианте SQL Server, поэтому источником несовместимости может оказаться и SQL Database.



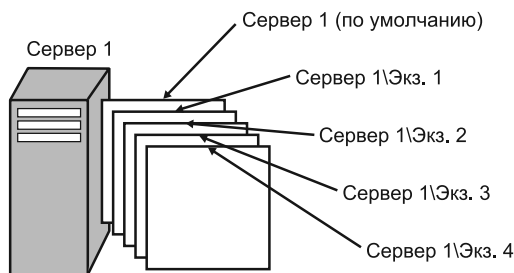
### ПРИМЕЧАНИЕ

Дополнение к книге, в котором описан процесс установки демонстрационной БД, рассчитан и на облачный сервис (на тот случай, если у вас есть к нему доступ).



## Экземпляры SQL Server

Как видно на рисунке 1.5, экземпляр SQL Server представляет собой отдельную установленную копию сервиса или ядра БД. На один компьютер можно установить несколько экземпляров локальной версии SQL Server, при этом каждый из них полностью автономен (в том числе с точки зрения безопасности и данных, за которые он отвечает). Физическое размещение экземпляров не принципиально — не важно, находятся они на одном сервере или в разных точках сети. Конечно, в первом случае экземпляры будут делить между собой одни и те же ресурсы (процессор, оперативную память, жесткий диск).



**Рис. 1.5.** Несколько экземпляров SQL Server на одном компьютере

Вы можете сделать так, чтобы один из экземпляров использовался по умолчанию, а остальные были доступны по имени. Это поведение определяется на этапе установки, позже его нельзя изменить. Чтобы подключиться к экземпляру по умолчанию, клиентское приложение должно указать доменное имя сервера или его IP-адрес. Для доступа к другому экземпляру в конце необходимо добавить обратный слеш (\) и указать имя (вводится при установке). Представьте, что на компьютере **Сервер1** установлены две копии SQL Server. Одна используется по умолчанию, а другая называется **Экз1**. В первом случае для подключения достаточно указать название сервера — **Сервер1**; во втором придется использовать запись **Сервер1\Экз1**.

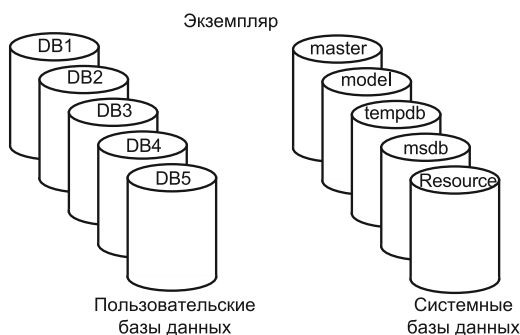
Рассмотрим ситуации, когда выгодно устанавливать несколько экземпляров SQL Server на один компьютер. Одна из причин — экономия на поддержке. Например, чтобы тестировать функциональность и поведение программного продукта, а также воспроизводить ошибки, возникающие в реальных условиях, в отделе технической поддержки должны находиться локальные копии SQL Server, которые имитируют пользовательскую среду, учитывая версии, разновидности и установленные обновления. Если организация использует несколько пользовательских окружений, отдел поддержки нуждается в соответствующих вариациях программного продукта. Вместо того чтобы выделять для этих задач множество серверов, каждый из которых обслуживает одну установленную копию, лучше использовать один компьютер с разными экземплярами SQL Server. Такого же результата можно достигнуть за счет запуска нескольких виртуальных машин.

В качестве другого примера можно взять таких людей, как я, которые занимаются преподаванием и читают лекций о языке T-SQL. Для нас возможность устанавливать несколько экземпляров SQL Server на один ноутбук крайне удобна. Это позволяет рассматривать разные версии продукта, демонстрируя отличие в их поведении и т. д.

Есть еще одна причина: компании, которые предоставляют БД в виде сервисов, должны гарантировать клиентам полную автономность и безопасность их информации. Раньше они предпочитали устанавливать несколько экземпляров SQL Server на один очень мощный компьютер. В последнее время облачные решения и усовершенствованные технологии виртуализации позволяют добиться аналогичных результатов.

## Базы данных

**База данных** — это контейнер для хранения таблиц, представлений, процедур и т. д. Как видно на рисунке 1.6, каждый экземпляр SQL Server может содержать несколько БД. При установке коробочной версии создается несколько системных БД, которые хранят информацию, необходимую для внутренних задач. У вас есть возможность создавать собственные БД для нужд программ.



**Рис. 1.6.** Пример нескольких баз данных в рамках одного экземпляра SQL Server

Установщик создает следующие системные БД: **master**, **Resource**, **model**, **tempdb** и **msdb**. Опишем каждую из них.

- **master**. Содержит метаданные об экземпляре SQL Server, конфигурацию сервера, сведения обо всех БД в текущем экземпляре, а также ресурсы, необходимые для инициализации.
- **Resource**. Скрытая БД, доступная только для чтения. Хранит определения всех системных объектов. Запрашивая системные объекты, обычно обращаются к схеме **sys** локальной БД, но на деле все определения находятся в БД **Resource**.
- **model**. Используется в качестве шаблона для новых БД. Каждая БД, которую вы создаете, изначально является копией **model**. Если хотите, чтобы новые БД были сконфигурированы определенным образом или содержали какой-то набор объектов (таких как типы данных), приведите к нужному виду БД **model**. Следует понимать, что изменения коснутся только новых БД, не затрагивая существующие.
- **tempdb**. Место, где SQL Server хранит временные данные (рабочие таблицы, пространство для сортировки, сведения о версиях строк и т. д.). Эта БД уничтожается и создается заново (на основе БД **model**) при каждом перезапуске экземпляра SQL Server.

- **msdb**. Здесь служба агента SQL Server хранит свои данные. Она отвечает за репликацию и автоматизацию задач, планирования и оповещения. **msdb** содержит сведения о таких компонентах и возможностях SQL Server, как Database Mail, Service Broker, механизм резервного копирования и т. д.

Локальный вариант SQL Server позволяет напрямую подключаться к БД **master**, **model**, **tempdb** и **msdb**. SQL Database дает прямой доступ только к **master**. При использовании временных таблиц или объявлении табличных переменных (подробнее об этом в главе 10) создается БД **tempdb**, но подключаться к ней и размещать внутри нее объекты вы не сможете.

В рамках одного экземпляра есть возможность создавать до 32 767 пользовательских БД для хранения объектов и программных ресурсов.

На уровне БД можно определить свойство под названием *collation*, которое отвечает за поддержку языка, чувствительность к регистру и порядок сортировки символьной информации. Если не указать это свойство при создании БД, для сравнения символов будет использоваться стандартная конфигурация.

Чтобы запустить код на языке T-SQL, клиент должен подключиться к экземпляру SQL Server и выбрать подходящую БД (то есть находиться в ее контексте).

С точки зрения безопасности, чтобы вы могли подключиться к экземпляру SQL Server, администратор БД должен создать для вас учетную запись. В локальной версии SQL Server учетную запись можно привязать к механизму аутентификации операционной системы Windows. В этом случае при подключении к SQL Server не нужно вводить логин и пароль, поскольку при входе в ОС вы автоматически становитесь авторизованным пользователем. Локальные и облачные варианты SQL Server также позволяют использовать полностью автономную систему аутентификации, которая требует ввода логина и пароля.

Администратору необходимо привязать вашу учетную запись к определенному пользователю в каждой БД, к которой вы хотите получить доступ. Пользователю БД выдаются права на работу с ее объектами.

SQL Server 2012 поддерживает изолированные БД, в которых действуют локальные, полностью автономные учетные записи, не относящиеся к серверу в целом. При создании таких записей администратор предоставляет пароль. Подключаясь к SQL Server, клиент указывает БД, с которой он хочет работать, логин и пароль; при этом он не может переключаться на другие пользовательские БД.

До этого момента мы рассматривали в основном логические аспекты баз данных. По большому счету, это единственное, что должно интересовать пользователей SQL Database, которые не сталкиваются с журнальными файлами, временными БД, физической структурой содержимого БД и т. д. Однако пользователи локальной версии SQL Server сами ответственны за физическую структуру. На рисунке 1.7 изображено устройство БД.

БД состоит из файлов, в которых хранятся как сами данные, так и журнал транзакций. При создании БД для каждого файла можно указывать различные свойства: местоположение, начальный и максимальный размеры, автоувеличение прироста размера. Оба вида файлов должны быть представлены как минимум в единственном экземпляре (в SQL Server это происходит по умолчанию). Файлы

данных содержат объекты, а в журнальные файлы записывается информация, необходимая для выполнения транзакций.

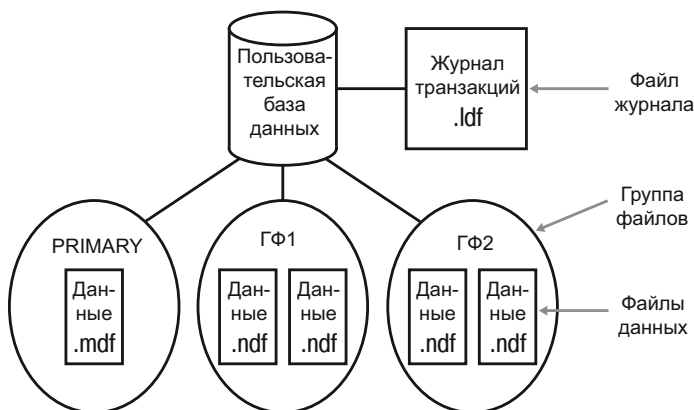


Рис. 1.7. Устройство базы данных

Хотя SQL Server способен вести запись сразу в несколько файлов данных, доступ к журнальным файлам всегда выполняется последовательно. Таким образом, разбиение журнала на несколько частей не улучшает производительности. Дополнительные журнальные файлы могут понадобиться в том случае, если на текущем разделе заканчивается свободное место.

Файлы данных организованы в виде групп, в рамках которых создаются различные объекты (таблицы или индексы). Объектные данные распространяются на все файлы, входящие в группу. С помощью этого механизма можно контролировать физическое размещение объектов. Любая БД содержит как минимум одну группу файлов под названием **PRIMARY**, в которой находятся первичный файл данных (с расширением **.mdf**) и системный каталог. В эту группу, как и в пользовательские, можно добавлять вторичные файлы данных (с расширениями **.ndf**). У вас есть возможность выбирать группу файлов по умолчанию; в ней будут создаваться объекты, для которых явно не указана какая-либо другая группа.



#### РАСШИРЕНИЯ ФАЙЛОВ .MDF, .LDF И .NDF

Все расширения файлов в БД имеют понятные значения. Расширение **.mdf** расшифровывается как Master Data File — главный файл данных (не путать с БД master), а **.ldf** означает Log Data File — файл журнала. Говорят, когда выбирали расширение для вторичных файлов данных, один из разработчиков в шутку предложил вариант **.ndf** (Not Master Data File — неглавный файл данных), и идею поддержали.

## Схемы и объекты

Я сознательно делал упрощение, когда говорил, что БД — это контейнер для хранения объектов. Как видно на рисунке 1.8, объекты содержатся внутри схем, которые в свою очередь находятся в базах данных. Схема — это и есть нечто вроде контейнера для таких объектов, как таблицы, представления, хранимые процедуры и т. д.

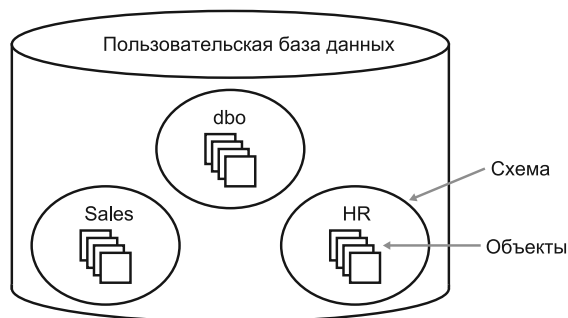


Рис. 1.8. База данных, ее схемы и объекты

Такой механизм позволяет управлять полномочиями. Например, вы можете выдать пользователю доступ к оператору `SELECT` внутри определенной схемы, чтобы он запрашивал данные из всех ее объектов. Безопасность — один из факторов, которые следует учитывать, разделяя объекты на схемы.

Схему также используют в качестве префиксов в названиях объектов. Представьте, что у вас есть таблица `Orders` внутри схемы `Sales`. Тогда полное название объекта выглядит как `Sales.Orders`. Если при работе с объектом имя схемы было опущено, SQL Server попытается определить его автоматически. Он сначала проверит, есть ли объект в схеме, которая используется по умолчанию, затем в схеме `dbo`. Компания Microsoft рекомендует всегда прописывать в коде полные названия объектов. Поиск не занимает много ресурсов, но зачем тратить их впустую? Кроме того, в разных схемах могут находиться объекты с одинаковыми именами, поэтому возможно, что в итоге вы получите совсем не то, на что рассчитывали.

## Создание таблиц и обеспечение целостности данных

Как я уже упоминал, главным предметом внимания в книге является язык DML. Однако вы все равно должны понимать, как создаются таблицы и определяется целостность данных. Я расскажу об этом, не углубляясь в подробности. Прежде чем перейти непосредственно к коду, напомним, что таблицы хранятся внутри схем, а схемы находятся в БД. В примерах, которые мы рассматриваем, используется демонстрационная БД `TSQL2012` и схема `dbo`.



### БОЛЬШЕ ИНФОРМАЦИИ

Если вы не знаете, как запускать код в SQL Server, вам поможет приложение в конце книги.

Схема `dbo` создается автоматически в каждой БД и используется по умолчанию для всех пользователей, которые не были явно привязаны к другой схеме.



### БОЛЬШЕ ИНФОРМАЦИИ

Информацию о создании демонстрационной БД вы найдете в приложении.

## Создание таблиц

Следующий код создает таблицу **Employees** в схеме **dbo**, которая в свою очередь находится внутри БД **TSQL2012**.

```
USE TSQL2012;

IF OBJECT_ID('dbo.Employees', 'U') IS NOT NULL
    DROP TABLE dbo.Employees;

CREATE TABLE dbo.Employees
(
    empid      INT NOT NULL,
    firstname  VARCHAR(30) NOT NULL,
    lastname   VARCHAR(30) NOT NULL,
    hiredate   DATE NOT NULL,
    mgrid      INT NOT NULL,
    ssn        VARCHAR(20) NOT NULL,
    salary     MONEY NOT NULL
);
```

Инструкция **USE** устанавливает контекст, то есть делает так, чтобы в качестве текущей базы данных использовалась **TSQL2012**. Благодаря этому скрипты создают объекты в определенной БД. В локальной версии **SQL Server** инструкция способна изменять текущий контекст. В **SQL Database** нельзя переключаться между БД, но если вы уже подключены к указанной БД, инструкция **USE** выполнится успешно. Я рекомендую использовать ее в любом случае, тогда ваши объекты всегда будут создаваться в нужной базе данных.

Оператор **IF** вызывает функцию **OBJECT\_ID**, чтобы проверить, существует ли таблица **Employees** в текущей БД. В качестве аргументов функция **OBJECT\_ID** принимает имя и тип объекта. Тип **'U'** представляет пользовательскую таблицу. Если ответ положительный, функция возвращает идентификатор объекта; в противном случае — значение **NULL**. В нашей ситуации код удаляет существующую таблицу и создает вместо нее новую. Естественно, можно предусмотреть и другую реакцию — например, оставить имеющийся объект.

Инструкция **CREATE TABLE** отвечает за определение того, что я называю заголовком отношения. Здесь нужно указать имя таблицы и в скобках набор ее атрибутов (столбцов).

Обратите внимание: я слеую собственным рекомендациям и указываю полное имя таблицы — **dbo.Employees**. Если опустить префикс, **SQL Server** будет считать, что пользователь запускает код в контексте схемы, применяемой по умолчанию в текущей БД.

Для каждого атрибута нужно указать имя, тип и свойство, которое определяет, может ли значение равняться **NULL** (то есть быть пустым).

В таблице **Employees** атрибуты **empid** (идентификатор служащего) и **mgrid** (идентификатор управляющего) имеют тип данных **INT** (целое число размером 4 байт); для атрибутов **firstname** (имя), **lastname** (фамилия) и **ssn** (номер социального страхования) выбран тип **VARCHAR** (строка переменной длины, для нее можно задать максимальное количество символов); атрибуты **hiredate** (день

приема на работу) и salary (зарплата) представлены типами DATE и MONEY соответственно.

Если явно не указать, должен ли столбец поддерживать значения NULL, SQL Server использует стандартную конфигурацию. Согласно спецификации языка SQL все столбцы по умолчанию могут быть пустыми (принимать отметки NULL), но в настройках это поведение можно изменить. Я настоятельно рекомендую вам явно определять такое свойство, не полагаясь на значения по умолчанию. Также советую всегда использовать запись NOT NULL и опускать ее только в случаях, когда возникает серьезная необходимость поддерживать значения NULL. Если не указать ограничение NOT NULL, в столбце появятся пустые значения. В таблице Employees NULL поддерживается только в столбце mgrid, ведь у сотрудника может не быть начальника — например, он сам является генеральным директором организации.



### СТИЛЬ ОФОРМЛЕНИЯ КОДА

Сделаю несколько замечаний относительно стиля оформления кода, использования точек с запятой, пробелов, табуляций, разделителей строк и т. д. Я не обращаю внимания на какие-либо формальные правила. Советую вам использовать тот стиль оформления кода, который удобен вашим коллегам-разработчикам. Главное, чтобы код был логичным, понятным и удобным в сопровождении. Я пытался приводить примеры, которые обладают вышеперечисленными свойствами.

В языке T-SQL пробельные символы можно использовать довольно свободно. С их помощью вы сделаете код более легким для восприятия. К примеру, в предыдущем разделе я мог записать весь код в одну строку, но многострочная версия с отступами выглядит понятнее.

Использование точки с запятой для разделения инструкций — стандартный подход. Кроме того, в некоторых системах БД это строгое требование. В SQL Server точка с запятой обязательна только в определенных местах, но никто не запрещает указывать ее и в других участках кода. Я рекомендую разделять с ее помощью все инструкции. Это не только улучшит удобочитаемость ваших запросов, но и в некоторых случаях убережет вас от проблем (SQL Server не всегда выводит понятное сообщение об ошибке, если пропустить точку с запятой там, где она необходима).



### ПРИМЕЧАНИЕ

В документации к SQL Server сказано, что возможность игнорировать точку с запятой, разделяя инструкции языка T-SQL, является устаревшей. Проще говоря, в будущих версиях продукта разработчики не смогут ей воспользоваться. Это еще один повод, чтобы начать разделять свои инструкции.

## Обеспечение целостности данных

Как упоминалось ранее, одно из прекрасных качеств реляционной модели — изначальная поддержка целостности данных. Целостность бывает декларативной (является частью определений таблиц) и процедурной (описывается внутри кода с помощью хранимых процедур или триггеров).

Тип данных, свойство, которое определяет возможность хранить пустые значения, и даже модель данных как таковая — примеры декларативного описания целостности данных. В этом разделе мы рассмотрим и другие ограничения подобного рода: первичные и внешние ключи, инструкции `UNIQUE`, `CHECK` и `DEFAULT`. Вы можете использовать данные ограничения, создавая таблицы (как часть выражения `CREATE TABLE`), или применять их к имеющимся таблицам в рамках конструкции `ALTER TABLE`. Все виды ограничений, кроме параметра `DEFAULT`, могут быть составными, то есть основываться сразу на нескольких атрибутах.

## Ограничения первичного ключа

Ограничение первичного ключа (`PRIMARY KEY`) обеспечивает уникальность строк и запрещает хранить значения `NULL` в соответствующих столбцах. Каждый уникальный набор значений в атрибутах, составляющих ограничение, может появиться в таблице только один раз — не более чем в одной строке. СУРБД пресекает любые попытки определить ограничения первичного ключа для столбцов, которые способны содержать пустые значения.

В примере, представленном ниже, данное ограничение определяется для атрибута `empid` из таблицы `Employees`.

```
ALTER TABLE dbo.Employees
    ADD CONSTRAINT PK_Employees
    PRIMARY KEY(empid);
```

Благодаря этому первичному ключу все значения атрибута `empid` будут уникальными и не пустыми. Операции добавления или обновления строк, которые нарушат описанное ограничение, отклонятся на уровне СУРБД и приведут к ошибке. Чтобы ограничение первичного ключа было однозначным, SQL Server автоматически создает уникальный индекс. Индексы (не обязательно уникальные) также помогают обойтись без полного сканирования таблиц, что значительно повышает скорость работы запросов (этот подход напоминает предметный указатель в книгах).

## Ограничение `UNIQUE`

Ограничение `UNIQUE` обеспечивает уникальность строк и реализует такую концепцию реляционной модели, как резервные ключи. В отличие от первичных ключей в одной таблице возможно присутствие нескольких ограничений `UNIQUE`. Кроме того, уникальность распространяется на все столбцы, включая те, которые могут иметь пустые значения. В соответствии со спецификацией SQL столбцы с ограничением `UNIQUE` поддерживают разные отметки типа `NULL` (как будто они отличаются). Однако язык T-SQL, реализованный в SQL Server, запрещает дублировать эти отметки (любые два значения `NULL` не могут быть равными).

Следующий код определяет ограничение `UNIQUE` для столбца `ssn` в таблице `Employees`.

```
ALTER TABLE dbo.Employees
    ADD CONSTRAINT UNQ_Employees_ssn
    UNIQUE(ssn);
```

Как и в случае с первичным ключом, в качестве механизма, который помогает соблюдать ограничения `UNIQUE`, используется уникальный индекс.



## Ограничения внешнего ключа

Внешний ключ отвечает за ссылочную целостность. Это ограничение определяется для одного или нескольких атрибутов в так называемой ссылающейся таблице и указывает на атрибуты потенциального ключа (ограничения PRIMARY KEY или UNIQUE) в таблице, на которую ссылаются (ее также называют ссылочной). Стоит заметить, что это может быть одна и та же таблица. Внешний ключ нужен для того, чтобы его атрибуты принимали только те значения, которые существуют в ссылочных столбцах.

Следующий код создает таблицу под названием **Orders**. Она содержит столбец **orderid** с первичным ключом.

```
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL
    DROP TABLE dbo.Orders;
CREATE TABLE dbo.Orders
(
    orderid INT          NOT NULL,
    empid   INT          NOT NULL,
    custid  VARCHAR(10)  NOT NULL,
    orderts DATETIME2    NOT NULL,
    qty     INT          NOT NULL,
    CONSTRAINT PK_Orders
        PRIMARY KEY(orderid)
);
```

Допустим, вы хотите ввести ограничение. Согласно ему столбец **empid** таблицы **Orders** должен поддерживать только те значения, которые находятся в столбце **empid** таблицы **Employees**. Чтобы добиться этого, необходимо определить ограничение внешнего ключа для столбца **empid** таблицы **Orders**, которое будет указывать на одноименный столбец таблицы **Employees**. Ниже показано, как это сделать.

```
ALTER TABLE dbo.Orders
    ADD CONSTRAINT FK_Orders_Employees
    FOREIGN KEY(empid)
    REFERENCES dbo.Employees(empid);
```

Аналогичным образом можно сослаться на столбец в той же таблице. С помощью следующего кода мы ограничим содержимое столбца **mgrid** значениями, которые хранятся в столбце **empid** (все это в таблице **Employees**).

```
ALTER TABLE dbo.Employees
    ADD CONSTRAINT FK_Employees_Employees
    FOREIGN KEY(mgrid)
    REFERENCES dbo.Employees(empid);
```

Обратите внимание: отметки NULL допускаются в столбцах внешнего ключа (в нашем случае **mgrid**), даже если их нет в столбцах, на которые они ссылаются.

В двух предыдущих примерах мы увидели простые определения внешнего ключа, способные вызывать ссылочное действие под названием NO ACTION. Это означает, что попытки удалить строку или обновить атрибуты потенциального ключа в таблице, на которую указывает ссылка, завершатся неудачей, если соответствующая строка

содержится в ссылающейся таблице. Например, если вы захотите удалить запись о сотруднике из таблицы **Employees** и при этом будут найдены соответствующие заказы в таблице **Orders**, СУРБД не даст вам этого сделать и вернет ошибку.

Вы можете определить внешний ключ с действиями, компенсирующими попытки удалить строку или обновить атрибуты потенциального ключа в таблице, на которую указывает внешняя ссылка. Речь идет о параметрах **ON DELETE** и **ON UPDATE**. В рамках определения внешнего ключа они поддерживают действия **CASCADE**, **SET DEFAULT** и **SET NULL**. Действие **CASCADE** приводит к тому, что выполняемые операции (удаление или обновление) будут распространяться и на связанные строки. Например, выражение **ON DELETE CASCADE** означает, что при удалении строки из таблицы, на которую указывает ссылка, СУРБД удалит также все связанные с ней строки из ссылающейся таблицы. Действия **SET DEFAULT** и **SET NULL** назначат атрибутам внешнего ключа в связанных строках соответственно значение по умолчанию и **NULL**. Независимо от выбранного действия указывающие «в никуда» строки могут появиться в ссылающейся таблице только в исключительном случае с отметками типа **NULL**.

## Ограничение CHECK

Ограничение **CHECK** позволяет определить предикат о том, что перед изменением или добавлением в таблицу строка должна пройти проверку на соответствие. Благодаря следующему ограничению столбец **salary** в таблице **Employees** поддерживает только положительные значения.

```
ALTER TABLE dbo.Employees
    ADD CONSTRAINT CHK_Employees_salary
    CHECK(salary > 0.00);
```

При попытке добавить или обновить строки СУРБД отклонит все значения, которые меньше или равны нулю. Обратите внимание: проверка считается не пройденной, если предикат возвращает **FALSE**. Если результат равен **TRUE** или **UNKNOWN**, изменения принимаются. Например, зарплата размером **-1000** будет отклонена, однако значения **50000** и **NULL** допустимы.

Используя инструкцию **CHECK** с ограничением внешнего ключа, вы можете указать параметр **WITH NOCHECK**, чтобы СУРБД не проводила проверку имеющихся данных. Это нежелательный подход, поскольку он не дает гарантий, что содержимое БД осталось последовательным. Можно также включать/отключать существующие инструкции **CHECK** и ограничения внешнего ключа.

## Ограничение DEFAULT

Ограничение **DEFAULT** связывается с определенным атрибутом. Выражение используется в качестве значения по умолчанию для пустых атрибутов строки, добавляемой в таблицу. В следующем примере ограничение **DEFAULT** определяется для столбца **orders** (он хранит время создания заказа).

```
ALTER TABLE dbo.Orders
    ADD CONSTRAINT DFT_Orders_orderts
    DEFAULT(SYSDATETIME()) FOR orderts;
```

Инструкция `DEFAULT` вызывает функцию `SYSDATETIME`, которая возвращает текущие дату и время. Теперь, если при добавлении записи в таблицу `Orders` вы не укажете определенное значение для столбца `orders`, SQL Server сделает это за вас.

В конце запустите приведенный код, чтобы очистить БД.

```
DROP TABLE dbo.Orders, dbo.Employees;
```

## В заключение

В этой главе мы ознакомились с прочным теоретическим фундаментом, лежащим в основе языка T-SQL. Начав с изучения архитектуры SQL Server, мы постепенно дошли до создания таблиц и обеспечения целостности данных. Надеюсь, теперь вы видите, что SQL — это нечто особенное, а не просто язык программирования.

## Глава 2

# ЗАПРОСЫ К ОДИНОЧНЫМ ТАБЛИЦАМ

В данной главе вы познакомитесь с принципами использования инструкции `SELECT`, позволяющей выполнять запросы к одиночным таблицам. Сперва мы рассмотрим структуру запроса; узнаем, из каких логических этапов он должен состоять, чтобы вернуть корректный результирующий набор. После перейдем к предикатам, операторам, выражениям `CASE`, отметкам `NULL`, одновременным операциям, извлечению метаданных, работе с символьной информацией, датой и временем. Многие примеры кода и упражнения используют БД `TSQL2012`. Инструкции по ее загрузке и установке вы найдете к книге.

## Элементы инструкции `SELECT`

Инструкция `SELECT` выполняет запрос к таблице, производит логические манипуляции и возвращает результат. В данном разделе речь пойдет о фазах, через которые проходит запрос. Вы узнаете о логическом порядке, в котором обрабатываются разные составляющие запроса, и о том, что происходит на каждом из этапов.

Говоря об этапах, я имею в виду принципиальный подход к обработке запроса и возвращению конечного результата, определенный на уровне спецификации языка `SQL`. Не удивляйтесь, если некоторые этапы покажутся вам неэффективными. Ядро `Microsoft SQL Server` не обязано в точности следовать данной логической цепочке обработки; оно может обращаться с запросом по-своему, меняя местами отдельные фазы, если это никак не повлияет на результат. Фактически, выполняя запросы, `SQL Server` делает множество разных оптимизаций.

Чтобы описать логическую цепочку обработки, а также операторы, которые могут быть в ней задействованы, используем запрос `SELECT` (листинг 2.1).

### Листинг 2.1. Пример запроса

```
USE TSQL2012;

SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
ORDER BY empid, orderyear;
```

Запрос отбирает заказы, которые сделал клиент под номером 71, затем группирует их по сотрудникам и годам. Остаются только те сотрудники и годы, для которых существует как минимум один заказ. Полученные группы содержат идентификатор

сотрудника, год заказа и их количество; сортировка производится по столбцам `empid` и `orderyear`. Пока не будем вникать в принцип работы запроса; сначала разобьем его на мелкие части и рассмотрим каждую инструкцию отдельно.

Код начинается с инструкции `USE`, которая переключает контекст вашей сессии на БД `TSQL2012`. Если вы уже работаете в этом контексте, инструкцию `USE` можно пропустить.

Прежде чем подробно рассматривать составляющие инструкции `SELECT`, остановимся на логическом порядке, в котором они обрабатываются. В большинстве языков программирования строчки кода выполняются в том порядке, в котором записаны. В `SQL` все иначе. Несмотря на то что инструкция `SELECT` находится в самом начале, обрабатываться она будет одной из самых последних. Выполнение операторов происходит в следующем порядке:

- 1) `FROM`;
- 2) `WHERE`;
- 3) `GROUP BY`;
- 4) `HAVING`;
- 5) `SELECT`;
- 6) `ORDER BY`.

Синтаксически запрос листинга 2.1 начинается с инструкции `SELECT`, но логически его составляющие размещены так:

```
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
ORDER BY empid, orderyear
```

Представим запрос в более понятной форме. Вот что делает это выражение:

- 1) запрашивает строки из таблицы `Sales.Orders`;
- 2) отбирает заказы, в которых идентификатор клиента равен 71;
- 3) группирует заказы по идентификатору сотрудника и году;
- 4) отбирает группы, которые содержат больше одного заказа;
- 5) возвращает для каждой группы идентификатор сотрудника, год заказа и их количество;
- 6) сортирует итоговые строки по идентификатору сотрудника и году заказа.

Запрос нельзя записывать в логическом порядке. Необходимо начинать с инструкции `SELECT`, как показано в листинге 2.1. Существует причина, по которой логический порядок обработки запроса и фактическая запись должны отличаться. `SQL` задумывался как декларативный язык программирования. Он позволяет оформлять запросы в виде выражений, похожих на английскую речь.

Рассмотрим команду, которую один человек дает другому: «Достань мне ключи от машины из левого верхнего ящика на кухне». Как видите, в начале команды упоминается объект, а потом указывается его местоположение. Однако если бы пришлось проинструктировать робота или компьютерную программу, вы бы начали с местоположения, а объект, который там можно найти, отметили в конце. Вот как бы это звучало: «Иди на кухню; открой левый верхний ящик; возьми ключи от машины; принеси их мне». Вводимые вами запросы похожи на разговорную речь — они начинаются с инструкции `SELECT` (то есть с действия, которое следует реализовать). Порядок логической обработки напоминает инструкции для робота — первым выполняется оператор `FROM` (то есть указывается место действия).

Усвоив порядок, в котором обрабатывается запрос, перейдем к подробному изучению каждой его составляющей.

Говоря о логической обработке запросов, я использую понятия «инструкции» и «этапы» (например, этап `WHERE` или инструкция `WHERE`). Инструкция — это синтаксическая составляющая запроса, поэтому, рассматривая синтаксис какого-либо элемента, я употребляю именно такой термин — «в инструкции `WHERE` указывается предикат». Анализируя логические манипуляции, которые выполняются в ходе обработки запроса, пользуюсь словами «этап» или «фаза» — «на этапе `WHERE` возвращаются строки, для которых предикат равен `TRUE`».

Вспомните мои советы из предыдущей главы об использовании точки с запятой для разделения инструкций. В настоящее время `SQL Server` требует это делать только в случаях, когда код может трактоваться двояко. Однако я рекомендую разделять таким образом все инструкции. Это сделает код более легким для восприятия и обеспечит совместимость на будущее, когда использование точки с запятой в `SQL Server` станет обязательным.

## Инструкция `FROM`

Именно с инструкции `FROM` начинается обработка. С ее помощью указываются имена таблиц, к которым следует выполнить запрос, и операторы, управляющие этими таблицами. Здесь мы не будем уделять большое внимание табличным операторам; подробнее о них поговорим в главах 3, 5 и 7. Пока воспринимайте инструкцию `FROM` как место, где задаются имена таблиц, с которыми вы хотите работать. Запрос, приведенный в листинге 2.1, подключается к таблице `Orders` внутри схемы `Sales` и находит 830 записей.

```
FROM Sales.Orders
```

В прошлой главе я советовал всегда использовать полные имена объектов, включая название схемы. Без схемы `SQL Server` будет искать объекты самостоятельно, что снижает производительность и может привести к выбору не того объекта.

Чтобы получить из таблицы все строки, не накладывая никаких условий, достаточно сделать запрос с инструкциями `FROM` и `WHERE`. В них необходимо указать соответственно имя нужной таблицы и интересующие вас атрибуты. К примеру, следующее выражение запрашивает все строки из таблицы `Orders` внутри схемы `Sales`, выбирая атрибуты `orderid`, `custid`, `empid`, `orderdate` и `freight`.

```
SELECT orderid, custid, empid, orderdate, freight  
FROM Sales.Orders;
```

Часть результата показана ниже.

orderid	custid	empid	orderdate	freight
-----	-----	-----	-----	-----
10248	85	5	2006-07-04 00:00:00.000	32,38
10249	79	6	2006-07-05 00:00:00.000	11,61
10250	34	4	2006-07-08 00:00:00.000	65,83
10251	84	3	2006-07-08 00:00:00.000	41,34
10252	76	4	2006-07-09 00:00:00.000	51,30
10253	34	3	2006-07-10 00:00:00.000	58,17
10254	14	5	2006-07-11 00:00:00.000	22,98
10255	68	9	2006-07-12 00:00:00.000	148,33
10256	88	3	2006-07-15 00:00:00.000	13,97
10257	35	4	2006-07-16 00:00:00.000	81,91
...				

(строк обработано: 830)

Может показаться, что результат запроса возвращается в определенном порядке, но никто не даст вам таких гарантий. Подробнее об этом я расскажу в разделах «Инструкция SELECT» и «Инструкция ORDER BY».



### ВЫДЕЛЕНИЕ ИМЕН ИДЕНТИФИКАТОРОВ

Если имена идентификаторов, которые используются в схемах, таблицах и столбцах, отвечают определенному формату, вам не нужно их выделять. Правила форматирования идентификаторов вы найдете в электронном справочнике для SQL Server по адресу [msdn.microsoft.com/ru-ru/library/ms175874](http://msdn.microsoft.com/ru-ru/library/ms175874). Если идентификатор оформлен неправильно (содержит пробелы или специальные символы, начинается с цифры или совпадает с зарезервированным ключевым словом), вы должны его выделить. В SQL Server это можно сделать несколькими способами. Обычно используются двойные кавычки (стандартная форма) — "Order Details". T-SQL также поддерживает квадратные скобки — [Order Details].

Например, таблицу с названием OrderDetails внутри схемы Sales можно записать как Sales.OrderDetails, "Sales"."OrderDetails" или [Sales].[OrderDetails]. Я предпочитаю не использовать выделение там, где без него можно обойтись. Если выбор идентификаторов зависит только от вас, рекомендую всегда соблюдать правила форматирования и указывать имена вроде OrderDetails, а не Order Details.

## Инструкция WHERE

Инструкция WHERE позволяет указать предикат или логическое выражение для фильтрации записей, возвращаемых на этапе FROM. До следующего этапа обработки дойдут только строки, для которых заданное логическое выражение окажется истинным (TRUE). В запросе, представленном в листинге 2.1, инструкция WHERE отбирала заказы, сделанные клиентом под номером 71.

```
FROM Sales.Orders
WHERE custid = 71
```

Из 830 строк, полученных после этапа FROM, операция WHERE отбирает 31. Чтобы увидеть записи, которые вернулись после применения фильтра `custid = 71`, запустите следующий запрос.

```
SELECT orderid, empid, orderdate, freight
FROM Sales.Orders
WHERE custid = 71;
```

Вот какой результат вы получите.

orderid	empid	orderdate	freight
10324	9	2006-10-08 00:00:00.000	214,27
10393	1	2006-12-25 00:00:00.000	126,56
10398	2	2006-12-30 00:00:00.000	89,16
10440	4	2007-02-10 00:00:00.000	86,53
10452	8	2007-02-20 00:00:00.000	140,26
10510	6	2007-04-18 00:00:00.000	367,63
10555	6	2007-06-02 00:00:00.000	252,49
10603	8	2007-07-18 00:00:00.000	48,77
10607	5	2007-07-22 00:00:00.000	200,24
10612	1	2007-07-28 00:00:00.000	544,08
10627	8	2007-08-11 00:00:00.000	107,46
10657	2	2007-09-04 00:00:00.000	352,69
10678	7	2007-09-23 00:00:00.000	388,98
10700	3	2007-10-10 00:00:00.000	65,10
10711	5	2007-10-21 00:00:00.000	52,41
10713	1	2007-10-22 00:00:00.000	167,05
10714	5	2007-10-22 00:00:00.000	24,49
10722	8	2007-10-29 00:00:00.000	74,58
10748	3	2007-11-20 00:00:00.000	232,55
10757	6	2007-11-27 00:00:00.000	8,19
10815	2	2008-01-05 00:00:00.000	14,62
10847	4	2008-01-22 00:00:00.000	487,57
10882	4	2008-02-11 00:00:00.000	23,10
10894	1	2008-02-18 00:00:00.000	116,13
10941	7	2008-03-11 00:00:00.000	400,81
10983	2	2008-03-27 00:00:00.000	657,54
10984	1	2008-03-30 00:00:00.000	211,22
11002	4	2008-04-06 00:00:00.000	141,16
11030	7	2008-04-17 00:00:00.000	830,75
11031	6	2008-04-17 00:00:00.000	227,22
11064	1	2008-05-01 00:00:00.000	30,09

(строк обработано: 31)

Инструкция WHERE играет большую роль с точки зрения производительности запросов. В зависимости от содержимого фильтра SQL Server может использовать индексы для доступа к нужным данным. В некоторых ситуациях индексы избавляют СУРБД от сканирования всей таблицы. Фильтры также снижают объем данных, передаваемых по сети, поскольку фильтрация происходит на стороне сервера, а клиенту передается конечный результат.

Как я упоминал, на этапе WHERE возвращаются только строки, которые удовлетворяют логическому выражению. Никогда не забывайте, что в языке T-SQL для



предикатов используется троичная логика. Она предусматривает три возможных результата: TRUE, FALSE и UNKNOWN. То есть «вернуть TRUE» — не то же самое, что «не вернуть FALSE». Инструкция WHERE опускает все записи, для которых логическое выражение является ложным (FALSE) или неопределенным (UNKNOWN). Подробнее об этом мы поговорим в разделе «Отметки NULL».

## Инструкция GROUP BY

Инструкция GROUP BY группирует строки, полученные на предыдущем этапе обработки. Группы определяются указанными вами элементами. Например, инструкция GROUP BY из листинга 2.1 содержит элементы empid и YEAR(orderdate).

```
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
```

Это означает, что инструкция GROUP BY создает группу для каждой уникальной комбинации (состоящей из идентификатора сотрудника и года), которая будет найдена в наборе после этапа WHERE. Выражение YEAR(orderdate) вызывает функцию YEAR, чтобы из даты в столбце orderdate возвращался только год.

На этапе WHERE мы получили 31 строку; 16 строк сформировали уникальные комбинации из идентификатора сотрудника и года. Ниже показан результат.

empid	YEAR(orderdate)
-----	-----
1	2006
1	2007
1	2008
2	2006
2	2007
2	2008
3	2007
4	2007
4	2008
5	2007
6	2007
6	2008
7	2007
7	2008
8	2007
9	2006

(строк обработано: 16)

Таким образом, каждая строка, возвращенная инструкцией WHERE, связана с одной из 16 групп, созданных на этапе GROUP BY.

Если данные в ходе запроса группируются, инструкции, которые следуют за этапом GROUP BY (включая HAVING, SELECT и ORDER BY), должны работать с группами, а не с отдельными строками. В итоге каждая группа будет представлена одной строкой. Это значит, что все выражения в инструкциях, следующих за оператором ORDER BY, должны возвращать для каждой группы скалярную величину (одиночное значение).

Выражения, основанные на элементах, которые указываются в инструкции GROUP BY, отвечают приведенным требованиям по определению, поскольку такие элементы могут входить в каждую группу только один раз. Например, в отдельно взятой группе существует лишь одно уникальное сочетание идентификатора сотрудника со значением 8 и заказа, сделанного в 2007 г. Таким образом, вы можете ссылаться на выражения empid и YEAR(orderdate) в инструкциях, которые следуют за этапом GROUP BY. Следующий запрос вернет 16 строк для 16 групп, состоящих из идентификатора сотрудника и года заказа.

```
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

Вот результат запроса.

empid	orderyear
1	2006
1	2007
1	2008
2	2006
2	2007
2	2008
3	2007
4	2007
4	2008
5	2007
6	2007
6	2008
7	2007
7	2008
8	2007
9	2006

(строк обработано: 16)

Элементы, не упоминающиеся на этапе GROUP BY, можно указывать только в агрегатных функциях (COUNT, SUM, AVG, MIN или MAX). Возьмем для примера запрос, который возвращает общую стоимость и количество заказов, сгруппированных по сотрудникам и годам.

```
SELECT
    empid,
    YEAR(orderdate) AS orderyear,
    SUM(freight) AS totalfreight,
    COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

Результат будет следующим.

empid	orderyear	totalfreight	numorders
1	2006	126,56	1

2	2006	89,16	1
9	2006	214,27	1
1	2007	711,13	2
2	2007	352,69	1
3	2007	297,65	2
4	2007	86,53	1
5	2007	277,14	3
6	2007	628,31	3
7	2007	388,98	1
8	2007	371,07	4
1	2008	357,44	3
2	2008	672,16	2
4	2008	651,83	3
6	2008	227,22	1
7	2008	1231,56	2

(строк обработано: 16)

Выражения `SUM(freight)` и `COUNT(*)` возвращают соответственно суммарную стоимость и количество (в нашем случае число строк) всех заказов в каждой группе. Если после группировки вы сошлетесь на атрибут, который отсутствует в инструкции `GROUP BY` (скажем, `freight`) и это происходит за пределами агрегатных функций, вы получите ошибку, ведь в таком случае нет гарантий, что выражение вернет для каждой группы только одно значение. К примеру, приведенный ниже запрос завершится неудачно.

```
SELECT empid, YEAR(orderdate) AS orderyear, freight
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate);
```

SQL Server выдаст следующую ошибку.

Сообщение 8120, уровень 16, состояние 1, строка 1

Столбец `"Sales.Orders.freight"` недопустим в списке выбора, поскольку он не содержится ни в агрегатной функции, ни в предложении `GROUP BY`.

Все агрегатные функции, за исключением `COUNT(*)`, игнорируют отметки типа `NULL`. Возьмем для примера группу из пяти строк со значениями 30, 10, `NULL`, 10, 10 в столбце под названием `qty`. Выражение `COUNT(*)` вернет 5, так как в группе находится пять записей. Однако результатом выражения `COUNT(qty)` будет 4, поскольку здесь учитываются только конкретные значения. Если вас интересуют лишь отдельные вхождения определенных результатов, укажите в скобках агрегатной функции ключевое слово `DISTINCT`. К примеру, выражение `COUNT(DISTINCT qty)` вернет 2, потому что у нас есть только два уникальных значения. Ключевое слово `DISTINCT` можно использовать и с другими функциями. Если выражение `SUM(qty)` возвращает 60, то результатом выполнения `SUM(DISTINCT qty)` будет 40. `AVG(qty)`, вернет 15, тогда как `AVG(DISTINCT qty)` выдаст 20. В качестве примера использования параметра `DISTINCT` в агрегатных функциях приведем

запрос, возвращающий количество разных клиентов, с которыми работал каждый сотрудник в каждом году.

```
SELECT
    empid,
    YEAR(orderdate) AS orderyear,
    COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY empid, YEAR(orderdate);
```

Запрос генерирует следующий результат.

empid	orderyear	numcusts
-----	-----	-----
1	2006	22
2	2006	15
3	2006	16
4	2006	26
5	2006	10
6	2006	15
7	2006	11
8	2006	19
9	2006	5
1	2007	40
2	2007	35
3	2007	46
4	2007	57
5	2007	13
6	2007	24
7	2007	30
8	2007	36
9	2007	16
1	2008	32
2	2008	34
3	2008	30
4	2008	33
5	2008	11
6	2008	17
7	2008	21
8	2008	23
9	2008	16

(строк обработано: 27)

## Инструкция HAVING

С помощью инструкции HAVING определяется предикат для фильтрации групп (напомню, что фильтрация отдельных строк происходит на этапе WHERE). На следующий этап обработки попадают только группы, для которых истинно логическое выражение в инструкции HAVING. Если предикат вернул FALSE или UNKNOWN, группы отсеиваются.

Поскольку инструкция HAVING выполняется после того, как записи сгруппированы, вы можете указывать в логическом выражении агрегатные функции. Например, в листинге 2.1 инструкция HAVING содержит предикат `COUNT (*) > 1`.

Он отбирает только группы (сотрудник и год заказа) с более чем одной строкой. В следующем фрагменте листинга 2.1 представлены этапы обработки, выполненные к этому моменту.

```
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
```

На этапе `GROUP BY` было создано 16 групп; семь из них содержат более одной строки, поэтому после выполнения инструкции `HAVING` остается всего девять групп. Убедитесь в этом, запустив приведенный запрос.

```
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
```

Результат показан ниже.

empid	orderyear
1	2007
3	2007
5	2007
6	2007
8	2007
1	2008
2	2008
4	2008
7	2008

(строк обработано: 9)

## Инструкция SELECT

В инструкции `SELECT` перечисляются атрибуты (столбцы), которые должен вернуть запрос. Выражения инструкции могут основываться как на атрибутах таблицы, так и на их производных. Например, в листинге 2.1 указаны следующие выражения: `empid`, `YEAR(orderdate)` и `COUNT(*)`. Название обычного атрибута совпадает с табличным. Однако вы можете присваивать атрибутам собственные имена; для этого существует инструкция `AS` (допустим, `empid AS employee_id`). Если выражения используют производные атрибутов, например `YEAR(orderdate)`, или вообще не основываются на исходных столбцах (как в случае с вызовом функции `CURRENT_TIMESTAMP`), их имена не попадают в конечный результат запроса (это можно изменить, назначив им специальные псевдонимы). И хотя язык T-SQL поддерживает возвращение безымянных столбцов, реляционная модель такого не допускает. Я настоятельно рекомендую использовать псевдонимы для выражений, подобных `YEAR(orderdate) AS orderyear`, чтобы у всех возвращаемых столбцов были имена. Тогда таблицу, полученную в результате запроса, можно считать реляционной.

Инструкция AS кажется наиболее понятной, но для полноты изложения я познакомлю вас и с другими способами создания псевдонимов. Итак, SQL Server поддерживает записи вида `<псевдоним> = <выражение>` («псевдоним равен выражению») и `<выражение> <псевдоним>` («выражение пробел псевдоним»). Примером первой записи служит фрагмент `orderyear = YEAR(orderdate)`, вторая будет выглядеть как `YEAR(orderdate) orderyear`. Форма, в которой выражение и псевдоним разделяются пробелом, довольно запутанна, поэтому лучше ее избегать.

Интересно, что, если случайно пропустить запятую между столбцами в инструкции SELECT, код не завершится ошибкой. SQL Server посчитает, что второе имя является псевдонимом для первого. Предположим, что при написании запроса, который отбирает столбцы `orderid` и `orderdate` из таблицы `Sales.Orders`, вы пропустили запятую между именами этих столбцов (как показано ниже).

```
SELECT orderid orderdate
FROM Sales.Orders;
```

С точки зрения синтаксиса запрос является корректным и подразумевает, что вы хотели создать для столбца `orderid` псевдоним под названием `orderdate`. В результате вы получите всего один столбец с именем `orderdate`, который хранит идентификаторы заказов.

```
orderdate
-----
10248
10249
10250
10251
10252
...
(строк обработано: 830)
```

Такую ошибку сложно распознать, поэтому следует быть предельно внимательным при написании кода.

Приведем инструкции из листинга 2.1, которые мы успели рассмотреть с учетом этапа SELECT.

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1
```

Инструкция SELECT формирует итоговую таблицу запроса. Заголовок нашей таблицы состоит из атрибутов `empid`, `orderyear` и `numorders`, а в теле находится девять строк (по одной на каждую группу). Чтобы получить эти строки, запустите следующий запрос.

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
```

Вот какой ответ будет сгенерирован.

empid	orderyear	numorders
1	2007	2
3	2007	2
5	2007	3
6	2007	3
8	2007	4
1	2008	3
2	2008	2
4	2008	3
7	2008	2

(строк обработано: 9)

Помните, что этап `SELECT` выполняется после обработки инструкций `FROM`, `WHERE`, `GROUP BY` и `HAVING`. Значит, псевдонимы, присвоенные выражениям в инструкции `SELECT`, на всех предыдущих этапах просто не существуют. Программистам, которые плохо знакомы с логическим порядком обработки запросов, свойственна одна и та же ошибка: они ссылаются на псевдонимы выражений в инструкции, предшествующих этапу `SELECT`. В качестве примера неправильного использования псевдонимов рассмотрим следующий запрос.

```
SELECTorderid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE orderyear > 2006;
```

На первый взгляд код корректен, но если помнить, что псевдонимы столбцов создаются на этапе `SELECT` (который выполняется после `WHERE`), становится понятно, что ссылка на `orderyear` в операторе `WHERE` недопустима. Это подтвердит и SQL Server, выдав ошибку.

```
Сообщение 207, уровень 16, состояние 1, строка 3
Недопустимое имя столбца "orderyear".
```

Данную проблему можно обойти несколькими способами. Например, указать выражение `YEAR(orderdate)` в обеих инструкциях — `WHERE` и `SELECT`.

```
SELECTorderid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE YEAR(orderdate) > 2006;
```

SQL Server способен определить повторное использование выражения, благодаря чему оно выполняется или вычисляется только один раз.

Следующий запрос — еще один пример неправильного использования псевдонимов столбцов. Он ссылается на псевдоним в инструкции `HAVING`, которая обрабатывается перед `WHERE`.

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING numorders > 1;
```

Запрос завершится ошибкой, в которой отражено, что имя столбца `numorders` указано неверно. Как и в предыдущем случае, выражение `COUNT(*)` должно находиться в обеих инструкциях.

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
WHERE custid = 71
GROUP BY empid, YEAR(orderdate)
HAVING COUNT(*) > 1;
```

В реляционной модели операции с отношениями основаны на реляционной алгебре; в результате их выполнения появляется новое отношение (множество). В языке SQL все немного иначе: операция `SELECT` не всегда возвращает настоящее множество, то есть набор уникальных неупорядоченных строк. В SQL таблица не обязана быть множеством. Уникальность строк не гарантируется без ключа, а это одно из обязательных свойств отношения в реляционной модели; без него получится мультимножество. Если таблицы, с которыми вы работаете, имеют ключи и являются множествами, результат выполнения инструкции `SELECT` все равно может содержать повторяющиеся строки. Такой итог часто называют **результующим набором**, и он не обязан соответствовать математическому определению множества. Например, таблица `Orders` является множеством, поскольку уникальность ее строк обеспечена ключом, однако запрос к ней возвращает повторяющиеся записи (листинг 2.2).

**Листинг 2.2.** Запрос, который возвращает повторяющиеся строки

```
SELECT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71;
```

Запрос генерирует следующий результат.

empid	orderyear
9	2006
1	2006
2	2006
4	2007
8	2007
6	2007
6	2007
8	2007
5	2007
1	2007
8	2007
2	2007
7	2007
3	2007
5	2007
1	2007
5	2007
8	2007
3	2007
6	2007



2	2008
4	2008
4	2008
1	2008
7	2008
2	2008
1	2008
4	2008
7	2008
6	2008
1	2008

(строк обработано: 31)

В языке SQL существуют средства, которые обеспечивают уникальность результата после выполнения инструкции `SELECT`. Речь идет о параметре `DISTINCT`. Он убирает из набора повторяющиеся строки (листинг 2.3).

**Листинг 2.3.** Запрос с параметром `DISTINCT`

```
SELECT DISTINCT empid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE custid = 71;
```

Результат выглядит так:

empid	orderyear
-----	-----
1	2006
1	2007
1	2008
2	2006
2	2007
2	2008
3	2007
4	2007
4	2008
5	2007
6	2007
6	2008
7	2007
7	2008
8	2007
9	2006

(строк обработано: 16)

Запрос, приведенный в листинге 2.2, вернул 31 строку; после удаления дубликатов, выполненного в листинге 2.3, осталось 16 строк.

Вместо списка отдельных атрибутов в инструкции `SELECT` можно указать символ звездочки (\*). Это позволит запросить сразу все столбцы соответствующей таблицы.

```
SELECT *
FROM Sales.Shippers;
```

В большинстве случаев такое применение звездочки — плохой метод. Даже если необходимо получить все атрибуты таблицы, каждый из них лучше указывать отдельно. Для этого существует множество причин. В отличие от реляционной модели язык SQL сохраняет порядок следования столбцов, который изначально использовался в выражении `CREATE TABLE`. Инструкция `SELECT *` всегда возвращает результат в этом порядке, игнорируя имена столбцов. Любые изменения в структуре таблицы (добавление или удаление атрибутов, изменение порядка их следования и т. д.) могут привести к ошибкам в клиентском приложении или к логическим ошибкам, которые останутся незамеченными. При явном перечислении нужных атрибутов всегда будет возвращаться ожидаемый результат (если указанные столбцы существуют в таблице). В случае удаления столбца, который упоминается в запросе, вы получите ошибку и сможете поправить код.

Возникает вопрос: существует ли разница в производительности между использованием звездочки и перечислением отдельных атрибутов? Когда указана звездочка, СУРБД тратит ресурсы, чтобы найти имена столбцов, хотя эти издержки обычно незаметны. Тем не менее явное перечисление столбцов является предпочтительным подходом, а выигрыш в скорости никогда не помешает.

В инструкции `SELECT` нельзя ссылаться на псевдоним столбца, который был создан в той же инструкции, и не важно, используется он справа или слева от места создания. Например, следующий код некорректен.

```
SELECT orderid,  
       YEAR(orderdate) AS orderyear,  
       orderyear + 1 AS nextyear  
FROM Sales.Orders;
```

Причины такого ограничения я объясню в разделе «Одновременные операции». Как уже говорилось, один из способов решения этой проблемы заключается в повторном использовании выражения.

```
SELECT orderid,  
       YEAR(orderdate) AS orderyear,  
       YEAR(orderdate) + 1 AS nextyear  
FROM Sales.Orders;
```

## Инструкция `ORDER BY`

Инструкция `ORDER BY` сортирует строки для их последующего вывода. С точки зрения логической обработки запроса данная операция выполняется в самом конце. В листинге 2.4 показан запрос, в котором строки сортируются по идентификатору сотрудника и году заказа.

### Листинг 2.4. Запрос, демонстрирующий работу инструкции `ORDER BY`

```
SELECT empid, YEAR(orderdate) AS orderyear, COUNT(*) AS numorders  
FROM Sales.Orders  
WHERE custid = 71  
GROUP BY empid, YEAR(orderdate)  
HAVING COUNT(*) > 1  
ORDER BY empid, orderyear;
```

Результат выглядит следующим образом.

empid	orderyear	numorders
-----	-----	-----
1	2007	2
1	2008	3
2	2008	2
3	2007	2
4	2008	3
5	2007	3
6	2007	3
7	2008	2
8	2007	4

(строк обработано: 9)

Теперь упорядоченность результатов гарантируется, чего нельзя сказать о запросах без инструкции `ORDER BY`.

Одна из важнейших особенностей языка SQL состоит в том, что таблица не гарантирует какой-либо упорядоченности, поскольку представляет собой множество (или мультимножество при наличии дубликатов), которое по определению не имеет порядка. Таким образом, если вы не воспользуетесь инструкцией `ORDER BY`, SQL Server сам решит, в каком порядке возвращать записи. Отсортировать строки в результирующем наборе можно исключительно с помощью инструкции `ORDER BY`. При этом результат нельзя называть таблицей. В спецификации языка SQL упорядоченные записи, которые возвращены подобным способом, называют **курсором** — нереляционным результатом с гарантированным порядком следования строк. Вам, наверное, интересно, почему важно, что мы получаем — таблицу или курсор. Некоторые операции и элементы языка SQL предназначены для работы только с табличными результатами; в качестве примеров можно привести табличные выражения и операторы работы с наборами (подробно о них рассказано в главах 5 и 6).

Обратите внимание: инструкция `ORDER BY` ссылается на псевдоним столбца `orderyear`, который был создан на этапе `SELECT`. Ранее я утверждал, что так делать нельзя, но инструкция `ORDER BY` — исключение, ведь она выполняется в последнюю очередь. Если вы определите псевдоним, название которого совпадает с именем исходного столбца (например, `col1 AS col1`), и сошлётесь на него в инструкции `ORDER BY`, для сортировки будет использован новый столбец.

Если вы хотите выполнить сортировку по возрастанию, укажите сразу после выражения параметр `ASC` (такой порядок сортировки используется по умолчанию) — `orderyear ASC`. Чтобы упорядочить строки по убыванию, примените параметр `DESC` — `orderyear DESC`.

Язык T-SQL позволяет ссылаться на столбцы в инструкции `ORDER BY` по их порядковому номеру. Он определяется размещением столбцов в списке атрибутов для инструкции `SELECT`. Например, в листинге 2.4 вместо

```
ORDER BY empid, orderyear
```

можно написать

```
ORDER BY 1, 2
```

Однако существует несколько причин, по которым такой подход считается нежелательным. Во-первых, в реляционной модели атрибуты никак не упорядочены, и к ним нужно обращаться по имени. Во-вторых, изменяя инструкцию `SELECT`, вы можете не внести соответствующие правки на этапе `ORDER BY`. Используя имена столбцов, вы защищаете код от подобных ошибок.

Язык T-SQL позволяет указывать в инструкции `ORDER BY` элементы, отсутствующие на этапе `SELECT`. Это означает, что вы можете выполнять сортировку по столбцам, которые не войдут в конечный результат. Следующий запрос сортирует записи о сотрудниках по дате найма, не возвращая атрибут `hiredate`.

```
SELECT empid, firstname, lastname, country
FROM HR.Employees
ORDER BY hiredate;
```

Однако, используя параметр `DISTINCT`, вы можете указывать в инструкции `ORDER BY` лишь элементы, которые присутствуют в списке `SELECT`. Причина ограничения заключается в том, что в результате работы параметра `DISTINCT` одна итоговая строка может представлять несколько исходных; поэтому SQL Server не будет знать, какое из возможных значений следует использовать на этапе `ORDER BY`. Рассмотрим некорректный запрос.

```
SELECT DISTINCT country
FROM HR.Employees
ORDER BY empid;
```

В таблице `Employees` содержатся записи о девяти сотрудниках: пять из них проживают в США, четыре — в Великобритании. Если опустить ошибочную инструкцию `ORDER BY`, запрос вернет две строки — по одной на каждую страну. Однако и США, и Великобритания встречаются в нескольких строках исходной таблицы и у каждой строки есть уникальный идентификатор сотрудника, поэтому значение выражения `ORDER BY empid` будет не определено.

## Фильтры TOP и OFFSET-FETCH

В этом разделе я расскажу о фильтрах, связанных с количеством строк и порядком их следования. Для начала рассмотрим инструкцию под названием `TOP`, которая поддерживается в SQL Server с версии 7.0. Затем познакомимся с новым фильтром, `OFFSET-FETCH`, появившимся в SQL Server 2012.

### Фильтр TOP

Параметр `TOP` — один из фирменных инструментов языка T-SQL. Он ограничивает количество строк (в абсолютных или относительных числах), возвращаемых запросом. Согласно спецификации у него есть два аргумента: первый содержит количество или процент строк, которые необходимо вернуть, а второй определяет порядок сортировки. Например, чтобы вернуть из таблицы `Orders` пять последних заказов, в инструкциях `SELECT` и `ORDER BY` следует указать выражения `TOP (5)` и `orderdate DESC` соответственно (листинг 2.5).

**Листинг 2.5.** Запрос, демонстрирующий работу фильтра TOP

```
SELECT TOP (5) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC
Вот результат запроса.
```

orderid	orderdate	custid	empid
11077	2008-05-06 00:00:00.000	65	1
11076	2008-05-06 00:00:00.000	9	4
11075	2008-05-06 00:00:00.000	68	8
11074	2008-05-06 00:00:00.000	73	7
11073	2008-05-05 00:00:00.000	58	2

(строк обработано: 5)

Как вы помните, инструкция ORDER BY вызывается после этапа SELECT, в котором указан параметр DISTINCT. То же справедливо и для фильтра TOP. Для фильтрации он использует инструкцию ORDER BY. То есть, если указать параметр DISTINCT на этапе SELECT, инструкция TOP будет выполнена после удаления повторяющихся строк.

При использовании фильтра TOP инструкция ORDER BY выполняет сразу две функции. Во-первых, определяет порядок, в котором в результате разместятся строки. Во-вторых, возвращает записи для фильтрации параметром TOP. Запрос, представленный в листинге 2.5, возвращает пять записей, которые содержат самые большие значения в столбце orderdate, и выводит их в порядке, указанном в выражении orderdate DESC.

Что же возвращает фильтр TOP — таблицу или курсор? Здесь легко запутаться. Обычно запросы с инструкцией ORDER BY выдают нереляционный результат (то есть курсор). Но что, если для получения реляционного результата фильтру TOP необходимо передать упорядоченные строки? И как относиться к ситуации, когда строки в инструкции TOP фильтруются в одном порядке, а выводятся уже в другом? Для этого используются табличные выражения. О них мы поговорим в главе 5.

Вы можете использовать инструкцию TOP в сочетании с ключевым словом PERCENT. В таком случае число возвращаемых столбцов будет вычисляться в процентах от общего количества с использованием округления. Следующий запрос возвращает 1 % от последних заказов.

```
SELECT TOP (1) PERCENT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Приведем результат запроса.

orderid	orderdate	custid	empid
11074	2008-05-06 00:00:00.000	73	7
11075	2008-05-06 00:00:00.000	68	8
11076	2008-05-06 00:00:00.000	9	4
11077	2008-05-06 00:00:00.000	65	1
11070	2008-05-05 00:00:00.000	44	2
11071	2008-05-05 00:00:00.000	46	1

11072	2008-05-05 00:00:00.000	20	4
11073	2008-05-05 00:00:00.000	58	2
11067	2008-05-04 00:00:00.000	17	1

(строк обработано: 9)

Запрос возвращает девять строк. Таблица **Orders** состоит из 830 записей. Один процент от 830 после округления равен девяти.

Вы, наверное, заметили, что список атрибутов в инструкции **ORDER BY** из листинга 2.5 не уникален: у столбца **orderdate** нет ни первичного ключа, ни ограничения **UNIQUE**. Несколько строк содержат одну и ту же дату заказа. Если не указать дополнительных условий, строки вернуться в неопределенном порядке. Данный запрос является **недетерминированным**, то есть имеет более одного корректного результата. При прочих равных строки будут упорядочены с учетом того, когда их использовали в первый раз. Фильтр **TOP** можно указывать в запросе, не содержащем инструкции **ORDER BY**. Тогда порядок следования записей будет полностью неопределенным — **SQL Server** вернет *n* первых строк, к которым осуществлялся физический доступ (где *n* — количество запрашиваемых записей).

Обратите внимание на результат листинга 2.5: самая старая дата заказа — 5 мая 2008 г. — указана только в одной строке. Такая дата может быть и в других строках таблицы, но мы не знаем, какая именно строка вернется, поскольку значения атрибутов в инструкции **ORDER BY** не уникальны.

Чтобы запрос был детерминированным, список атрибутов в инструкции **ORDER BY** следует сделать уникальным; проще говоря, необходимо добавить дополнительное условие. Например, вы можете указать выражение **orderid DESC**, как показано в листинге 2.6. В таком случае при прочих равных предпочтение будет отдано записи с большим порядковым номером.

**Листинг 2.6.** Фильтр **TOP** и инструкция **ORDER BY** с уникальным списком атрибутов

```
SELECT TOP (5) orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC, orderid DESC;
Запрос вернет следующий результат.
```

orderid	orderdate	custid	empid
-----	-----	-----	-----
11077	2008-05-06 00:00:00.000	65	1
11076	2008-05-06 00:00:00.000	9	4
11075	2008-05-06 00:00:00.000	68	8
11074	2008-05-06 00:00:00.000	73	7
11073	2008-05-05 00:00:00.000	58	2

(строк обработано: 5)

Результаты листингов 2.5 и 2.6 ничем не отличаются. Разница лишь в том, что вывод листинга 2.5 является одним из нескольких возможных, а у запроса, представленного в листинге 2.6, может быть только один корректный результат.

Вместо добавления дополнительного условия в список элементов инструкции **ORDER BY** вы можете сделать так, чтобы возвращались все равнозначные

записи. Например, помимо пяти строк, которые вернул запрос в листинге 2.5, в таблице содержится еще несколько записей с тем же сортировочным значением (в нашем случае это дата заказа), что и у последней найденной записи (5 мая 2008 г.). Чтобы их вернуть, добавьте параметр `WITH TIES`, как показано в следующем запросе.

```
SELECT TOP (5) WITH TIES orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate DESC;
```

Результат представлен ниже.

orderid	orderdate	custid	empid
11077	2008-05-06 00:00:00.000	65	1
11076	2008-05-06 00:00:00.000	9	4
11075	2008-05-06 00:00:00.000	68	8
11074	2008-05-06 00:00:00.000	73	7
11073	2008-05-05 00:00:00.000	58	2
11072	2008-05-05 00:00:00.000	20	4
11071	2008-05-05 00:00:00.000	46	1
11070	2008-05-05 00:00:00.000	44	2

(строк обработано: 8)

Обратите внимание: несмотря на то что мы указали выражение `TOP (5)`, результат состоит из восьми строк. SQL Server возвращает первые пять записей, исходя из порядка сортировки `orderdate DESC`, и добавляет к ним остальные строки с тем же значением атрибута `orderdate`, что и у записи, доступ к которой осуществлялся раньше всех.

## Фильтр OFFSET-FETCH

Фильтр `TOP` очень практичный, но обладает двумя недостатками: он входит в стандарт и не поддерживает пропуск отдельных результатов. В спецификации языка SQL содержится аналогичный фильтр под названием `OFFSET-FETCH`; у него нет перечисленных проблем, что позволяет разбивать результат на произвольное количество страниц. Поддержка фильтра реализована в SQL Server 2012.

В T-SQL фильтр `OFFSET-FETCH` является частью этапа `ORDER BY`, на котором результат сортируется для последующего вывода. С помощью инструкции `OFFSET` можно указать, сколько строк необходимо пропустить, а инструкция `FETCH` позволяет определить количество фильтруемых записей, начиная с пропущенных. Рассмотрим следующий запрос.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;
```

Здесь строки в таблице `Orders` упорядочиваются по столбцам `orderdate` и `orderid` (по возрастанию; столбец `orderid` выступает в качестве дополнительного условия). Исходя из такой сортировки, инструкция `OFFSET` пропустит 50 первых строк, после чего инструкция `FETCH` отберет следующих 25.

Запрос, в котором используется фильтр `OFFSET-FETCH`, должен содержать инструкцию `ORDER BY`. Кроме того, инструкцию `FETCH` нельзя использовать без `OFFSET`. Чтобы применить фильтр, не пропуская ни одной строки, необходимо использовать выражение `OFFSET 0 ROWS`. Однако инструкция `OFFSET` работает и без `FETCH`. В этом случае запрос пропустит определенное количество строк и вернет все, что останется.

У языка `T-SQL` есть одна интересная особенность, которая касается синтаксиса фильтра `OFFSET-FETCH`: параметры `ROW` и `ROWS` являются взаимозаменяемыми. Они придают запросу сходство с английским языком, позволяя выбирать между единственным и множественным числом. Представьте, что вы хотите извлечь только одну строку. Выражение `FETCH 1 ROWS` синтаксически корректно, но с точки зрения английской грамматики выглядит несуразным. Поэтому вы можете написать `FETCH 1 ROW`. То же относится к инструкции `OFFSET`. Кроме того, если не пропускается ни одной строки (`OFFSET 0 ROWS`), вместо `NEXT` (следующих) есть возможность указать `FIRST` (первых) — эти ключевые слова также взаимозаменяемы.

Фильтр `OFFSET-FETCH` более гибкий, чем `TOP`, так как умеет пропускать записи. И хотя в отличие от `TOP` он не поддерживает параметры `PERCENT` и `WITH TIES`, я рекомендую использовать именно его. При необходимости вы всегда сможете применить специфические возможности фильтра `TOP`.

## Краткий обзор оконных функций

**Оконными** называют функции, которые на основе окна (набора строк) вычисляют **скалярное** (одиночное) значение. Окно определяется с помощью инструкции `OVER`. Такие функции крайне практичны. `SQL Server` поддерживает несколько видов оконных функций. Однако на данном этапе преждевременно углубляться в столь интересную тему. Сейчас я попробую донести основную идею на примере функции `ROW_NUMBER`. В главе 7 вы получите больше подробностей.

Оконные функции работают с наборами строк, которые предоставляются инструкцией под названием `OVER`. Эта инструкция передает в функцию часть набора, полученного в результате исходного запроса. Вы можете сузить набор строк в окне с помощью вложенной инструкции `PARTITION BY`, а также указать порядок вычисления (если он имеет значение), используя инструкцию `ORDER BY` (не путать с этапом `ORDER BY`, который выполняется на уровне запроса).

Рассмотрим следующий код.

```
SELECT orderid, custid, val,
       ROW_NUMBER() OVER(PARTITION BY custid
                          ORDER BY val) AS rownum
FROM Sales.OrderValues
ORDER BY custid, val;
```

Вот какой результат он генерирует.

orderid	custid	val	rownum
10702	1	330.00	1
10952	1	471.20	2
10643	1	814.50	3



---

10835	1	845.80	4
10692	1	878.00	5
11011	1	933.50	6
10308	2	88.80	1
10759	2	320.00	2
10625	2	479.75	3
10926	2	514.40	4
10682	3	375.50	1
...			
(строк обработано: 830)			

Функция `ROW_NUMBER` присваивает итоговым строкам уникальные последовательные целые значения в рамках соответствующих разделов, основываясь на фактическом порядке их следования. Инструкция `OVER` делит окно, исходя из значений атрибута `custid`, в результате чего каждый клиент получает уникальные номера строк. Инструкция `OVER` также упорядочивает содержимое окна по атрибуту `val`, поэтому последовательные номера строк инкрементируются в рамках разделов, основанных на данном атрибуте.

Функция `ROW_NUMBER` должна выдавать уникальные значения в рамках каждого раздела. Таким образом, даже если значение, по которому выполнялась сортировка, остается прежним, номер строки все равно должен увеличиваться. Если вложенная инструкция `ORDER BY` внутри функции `ROW_NUMBER` содержит не уникальный список атрибутов, запрос получится недетерминированным. Чтобы сделать вычисление номера строки четко определенным, добавьте в список `ORDER BY` дополнительные элементы (например, атрибут `orderid`).

Как уже упоминалось, инструкция `ORDER BY`, отраженная внутри `OVER`, не имеет отношения к конечному выводу и не влияет на реляционную природу результата. Если вы не укажете инструкцию `ORDER BY` на уровне запроса, порядок следования строк в результирующем наборе будет неопределенным.

Не забывайте, что выражения в списке `SELECT` обрабатываются перед выполнением инструкции `DISTINCT` (если она имеется). Это относится и к выражениям, основанным на оконных функциях. О значимости данного факта я расскажу в главе 7.

Итак, чтобы подытожить вышесказанное, приведу порядок обработки рассмотренных нами инструкций:

- 1) `FROM`;
  - 2) `WHERE`;
  - 3) `GROUP BY`;
  - 4) `HAVING`;
  - 5) `SELECT`;
  - 6) выражения;
  - 7) `DISTINCT`;
  - 8) `ORDER BY`;
  - 9) `TOP / OFFSET-FETCH`.
-

## Предикаты и операторы

Язык T-SQL содержит элементы, в рамках которых можно задавать предикаты, — это фильтры запросов вроде WHERE или ограничения HAVING, CHECK и т. д. Как вы помните, предикаты являются логическими выражениями, которые возвращают одно из трех значений: TRUE, FALSE или UNKNOWN. Для объединения предикатов предусмотрены логические операторы AND и OR. Вы можете использовать и другие конструкции, например операторы сравнения.

T-SQL поддерживает различные предикаты, включая IN, BETWEEN и LIKE. Предикат IN проверяет, равно ли значение (или скалярное выражение) как минимум одному элементу множества. Приведенный ниже запрос возвращает заказы с идентификаторами 10248, 10249 и 10250.

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderid IN(10248, 10249, 10250);
```

Предикат BETWEEN определяет, находится ли значение в диапазоне, заданном с помощью двух граничных значений. Следующий запрос возвращает заказы в диапазоне от 10300 включительно до 10310 включительно.

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderid BETWEEN 10300 AND 10310;
```

Предикат LIKE анализирует, соответствует ли строковое значение заданному шаблону. Например, следующий запрос возвращает записи о сотрудниках, чьи фамилии начинаются с буквы «Д».

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname LIKE N'Д%';
```

Подробнее о поиске по шаблону и предикате LIKE мы поговорим позже в этой главе.

Обратите внимание: перед строкой 'Д%' указан префикс N; это сокращение от слова «national» («национальный»). Он используется в тех случаях, когда строка состоит не из обычных символов (CHAR или VARCHAR), а имеет тип Unicode (NCHAR или NVARCHAR). Мы указали префикс, потому что атрибут lastname имеет тип NVARCHAR(40). О том, как обращаться со строками, я расскажу в разделе «Работа с символьными данными».

T-SQL поддерживает следующие операторы сравнения: =, >, <, >=, <=, <>, !=, !>, !<. Три последних не входят в спецификацию языка SQL, поэтому я не рекомендую их использовать. Кроме того, у каждого из них есть стандартные альтернативы (например, <> вместо !=). Следующий запрос возвращает все заказы, сделанные с 1 января 2008 г.

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20080101';
```

Для объединения логических выражений используются логические операторы OR и AND. Оператор NOT служит для инвертирования. Приведенный ниже запрос возвращает заказы, размещенные после 1 января 2008 г. и обработанные сотрудниками с идентификаторами 1, 3 или 5.

```
SELECT orderid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20080101'
      AND empid IN(1, 3, 5);
```

В состав языка T-SQL входят стандартные арифметические операторы +, -, \* и /, а также оператор %, который возвращает остаток деления целых чисел. Следующий запрос вычисляет чистую стоимость на основе атрибутов quantity (количество), unitprice (цена за единицу товара) и discount (скидка).

```
SELECT orderid, productid, qty, unitprice, discount,
       qty * unitprice * (1 - discount) AS val
FROM Sales.OrderDetails;
```

Тип скалярного выражения, в котором участвуют два операнда, определяется на основе приоритета типов данных. Если операнды имеют один тип данных, тип результата будет соответствующим. Например, при делении двух целых чисел (INT) получается целое число. Выражение 5/2 возвращает 2, а не 2,5. При работе с константами это не вызывает проблем, потому что вы всегда можете представить значение в виде числа с запятой. Однако если вы делите содержимое двух целочисленных столбцов (скажем, col1/col2), необходимо привести операнды к подходящим типам, чтобы в результате получить действительное число: CAST(col1 AS NUMERIC(12, 2))/CAST(col2 AS NUMERIC(12, 2)). Тип данных NUMERIC(12, 2) имеет точность 12 и масштаб 2, то есть он состоит из 12 цифр, две из которых находятся после запятой.

Если операнды имеют разные типы, выражение сводится к типу с большим приоритетом. Например, в выражении 5/2.0 первый операнд целочисленный (INT), а второй представлен числом с плавающей запятой (NUMERIC). Поскольку у типа NUMERIC более высокий приоритет по сравнению с INT, операнд 5 перед математической операцией автоматически превращается в действительное число (5,0). В итоге вы получите 2,5.

Перечень типов и их приоритеты вы найдете в электронном справочнике в разделе «Приоритет типов данных (Transact-SQL)».

SQL Server руководствуется аналогичным подходом и в ситуациях, когда в одном выражении задействовано несколько операторов. Приведем список операторов в порядке убывания приоритета:

- 1) ( ) (скобки);
- 2) \* (умножение), / (деление), % (остаток);
- 3) + (положительное число), - (отрицательное число), + (добавление), + (сложение), - (вычитание);
- 4) =, >, <, >=, <=, <>, !=, !>, !< (операторы сравнения);
- 5) NOT;

- 6) AND;
- 7) BETWEEN, IN, LIKE, OR;
- 8) = (присваивание).

В следующем запросе оператор AND имеет приоритет перед OR.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE
    custid = 1
    AND empid IN(1, 3, 5)
    OR custid = 85
    AND empid IN(2, 4, 6);
```

Запрос возвращает заказы двух категорий: «размещенные клиентом 1 и обработанные сотрудниками 1, 3 или 5» и «размещенные клиентом 85 и обработанные сотрудниками 2, 4 или 6».

Скобки обладают наивысшим приоритетом, поэтому с их помощью можно получить полный контроль над процессом выполнения запроса. Их рекомендуется использовать даже там, где они не обязательны; это поможет другим изучать и поддерживать ваш код и сделает запросы более разборчивыми. То же касается отступов. К примеру, следующий запрос не отличается от предыдущего, но разобраться в нем намного легче.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE
    (custid = 1
     AND empid IN(1, 3, 5))
    OR
    (custid = 85
     AND empid IN(2, 4, 6));
```

С помощью скобок можно управлять приоритетом не только логических, но и арифметических операторов. В приведенном ниже коде умножение выполняется перед сложением.

```
SELECT 10 + 2 * 3;
```

Таким образом, мы получим 16. Если поставить скобки, сначала будет выполняться сложение.

```
SELECT (10 + 2) * 3;
```

На этот раз выражение вернет 36.

## Выражение CASE

Выражение CASE является скалярным и возвращает значение на основе условной логики. Стоит отметить, что это не предложение; то есть оно не позволит вам

управлять ходом выполнения запроса или вводить дополнительные условия. Для него действуют те же правила, что и для других скалярных выражений, таких как ограничение CHECK или инструкции SELECT, WHERE, HAVING и ORDER BY.

У выражения CASE есть две формы: простая и поисковая. Простая форма сравнивает одно значение или скалярное выражение со списком возможных вариантов, возвращая первое совпадение. Если ни один из элементов списка не совпал с заданным значением, выражение CASE возвращает то, что указано в инструкции ELSE. Если инструкция ELSE отсутствует, возвращается NULL.

Следующий код выполняет запрос к таблице **Production.Products**. В инструкции SELECT используется выражение CASE. С его помощью описываются значения в столбце **categoryid**.

```
SELECT productid, productname, categoryid,
CASE categoryid
  WHEN 1 THEN 'Напитки'
  WHEN 2 THEN 'Приправы'
  WHEN 3 THEN 'Сладости'
  WHEN 4 THEN 'Молочные продукты'
  WHEN 5 THEN 'Зерно/крупы'
  WHEN 6 THEN 'Мясо/птица'
  WHEN 7 THEN 'Овощи'
  WHEN 8 THEN 'Морепродукты'
  ELSE 'Категория не определена'
END AS categoryname
FROM Production.Products;
```

Часть результата запроса представлена ниже.

productid	productname	categoryid	categoryname
1	Product HHYDP	1	Напитки
2	Product RECZE	1	Напитки
3	Product IMENJ	2	Приправы
4	Product KSBRM	2	Приправы
5	Product EPEIM	2	Приправы
6	Product VAIIV	2	Приправы
7	Product HMLNI	7	Овощи
8	Product WVJFP	2	Приправы
9	Product AOZBW	6	Мясо/птица
10	Product YHXGE	8	Морепродукты
...			

(строк обработано: 77)

Предыдущий запрос — пример простой формы выражения CASE. Если речь идет о более объемном и динамичном списке продуктовых категорий, логичнее выделять для выражения отдельную таблицу и затем соединить ее с таблицей **Products** во время запроса. В БД **TSQL2012** такая таблица существует — она называется **Categories**.

В правой части простой формы выражения CASE находится единственное проверочное значение, которое сравнивается со списком вариантов, обозначенных инструкциями WHEN. Поисковая форма CASE обладает большей гибкостью; она определяет в инструкциях WHEN предикаты или логические выражения, не ограничиваясь обычной проверкой на равенство. Значение возвращается ин-

струкцией THEN, связанной с первым логическим выражением WHEN, результат которого равен TRUE. Если ни одно из выражений не выдало TRUE, возвращается значение, которое указано в инструкции ELSE (или NULL, если такой инструкции нет). Следующий запрос описывает три ценовые категории: до 1000, от 1000 до 3000 и больше 3000.

```
SELECT orderid, custid, val,
       CASE
         WHEN val < 1000.00                THEN 'Меньше 1000'
         WHEN val BETWEEN 1000.00 AND 3000.00 THEN 'От 1000 до 3000'
         WHEN val > 3000.00                THEN 'Больше 3000'
         ELSE 'Неизвестно'
       END AS valuecategory
FROM Sales.OrderValues;
```

Вот какой результат генерируется.

orderid	custid	val	valuecategory
10248	85	440.00	Меньше 1000
10249	79	1863.40	От 1000 до 3000
10250	34	1552.60	От 1000 до 3000
10251	84	654.06	Меньше 1000
10252	76	3597.90	Больше 3000
10253	34	1444.80	От 1000 до 3000
10254	14	556.62	Меньше 1000
10255	68	2490.50	От 1000 до 3000
10256	88	517.80	Меньше 1000
10257	35	1119.90	От 1000 до 3000
...			

(строк обработано: 830)

Как видите, любую простую форму выражения CASE легко превратить в поисковую, однако обратное утверждение не всегда справедливо.

В языке T-SQL есть несколько функций, которые можно считать урезанными разновидностями выражения CASE: ISNULL, COALESCE, IIF и CHOOSE. Только COALESCE является стандартной. IIF и CHOOSE доступны исключительно в SQL Server 2012.

Функция ISNULL принимает два аргумента, возвращая тот, который первым идентифицируется как «не пустой» (не равен NULL); если оба аргумента равны NULL, возвращается NULL. Например, выражение ISNULL(col1, '') вернет col1, если значение не равно NULL; в противном случае результатом будет пустая строка. Функция COALESCE работает аналогичным образом, только ей можно передавать от более двух аргументов. Я рекомендую использовать стандартные элементы языка, то есть вместо ISNULL лучше выбрать функцию COALESCE.

Нестандартные функции IIF и CHOOSE добавили в SQL Server 2012, чтобы облегчить процесс перехода с Microsoft Access. Функция IIF (<логическое\_выражение>, <выр1>, <выр2>) возвращает выр1, если логическое\_выражение истинно; иначе возвращается выр2. Например, результатом IIF(col1 <> 0, col2/col1, NULL) будет col2/col1, если col1 не равно 0; в противном случае мы получим NULL. Функция CHOOSE (<индекс>, <выр1>, <выр2>, ..., <вырп>) возвращает выражение из списка по заданному индексу.

Например, `CHOOSE(3, col1, col2, col3)` вернет значение `col3`. В функции `CHOOSE` обычно используются более динамичные выражения, которые могут зависеть от данных, введенных пользователем.

Итак, мы познакомились с выражением `CASE` и несколькими функциями с похожим назначением. На данный момент нам удалось рассмотреть только простейшие примеры, но имейте в виду, что это чрезвычайно мощный и полезный элемент языка.

## Отметки NULL

Как говорилось в первой главе, для поддержки отсутствующих значений и троичной логики в языке `SQL` используются отметки `NULL`; другими словами, предикаты возвращают значения `TRUE`, `FALSE` или `UNKNOWN`. В этом отношении `T-SQL` соответствует стандарту. Работа с отметками `NULL` и значениями `UNKNOWN` кажется довольно запутанной, поскольку на интуитивном уровне люди склонны мыслить терминами двоичной логики (`TRUE` и `FALSE`). К тому же некоторые элементы языка `SQL` обращаются со значениями `NULL` и `UNKNOWN` по-разному.

Начнем с исчисления предикатов в троичной системе. Логическое выражение, которое проверяет существование чего-либо или просто выводит данные, всегда возвращает либо `TRUE`, либо `FALSE`. Если используется отсутствующее значение, результат получится неопределенным (`UNKNOWN`). Возьмем для примера предикат `salary > 0`. Если `salary` равно 1000, выражение возвращает `TRUE`. Если `salary` равно -1000, выражение возвращает `FALSE`. Когда внутри `salary` хранится отметка `NULL`, мы получим `UNKNOWN`.

В `SQL` значения `TRUE` и `FALSE` ведут себя довольно прозрачно. Если фильтр (размещенный в инструкциях `WHERE` или `HAVING`) содержит предикат `salary > 0`, запрос вернет строки, для которых выражение равно `TRUE`; если предикат возвращает `FALSE`, строки отсеиваются. Аналогичным образом, если предикат `salary > 0` указан в ограничении `CHECK`, предложения `INSERT` и `UPDATE` будут действовать на строки, для которых предикат возвращает `TRUE`; при возврате `FALSE` строки отклоняются.

Значения `UNKNOWN` обрабатываются по-разному в зависимости от контекста. Например, фильтры в языке `SQL` принимают только `TRUE`, то есть значения `FALSE` и `UNKNOWN` всегда отсеиваются. Ограничение `CHECK`, наоборот, отклоняет только `FALSE`, в результате принимается не только значение `TRUE`, но и `UNKNOWN`. В двоичной системе исчисления предикатов между «принимать `TRUE`» и «отклонять `FALSE`» нет разницы. Однако мы работаем с троичной системой, поэтому предикат «принимать `TRUE`» отсеивает как `UNKNOWN`, так и `FALSE`, а выражение «отклонять `FALSE`» подразумевает, что `UNKNOWN` проходит проверку наравне с `TRUE`. Предикат `salary > 0` из предыдущего примера вернет `UNKNOWN`, если `salary` равно `NULL`. Если поместить такой предикат в инструкцию `WHERE`, все строки, у которых `salary` равняется `NULL`, отсеются. Однако в ограничении `CHECK` он будет означать, что пустые строки должны приниматься.

У значения `UNKNOWN` есть ключевая особенность: оно не меняется в результате отрицания. Возьмем предикат `NOT (salary > 0)`: если `salary` равно `NULL`, выражение `salary > 0` вернет `UNKNOWN`, а `NOT UNKNOWN` также равняется `UNKNOWN`.

Некоторые считают странным, что операция сравнения двух пустых значений (`NULL = NULL`) возвращает `UNKNOWN`. С точки зрения языка SQL отметка `NULL` представляет сущность, которая отсутствует либо не определена. Нельзя сказать с уверенностью, равны ли два неизвестных значения. В связи с этим в SQL предусмотрено два предиката — `IS NULL` и `IS NOT NULL`, которые необходимо использовать вместо выражений `= NULL` и `<> NULL`.

Сейчас я на практике продемонстрирую аспекты троичной системы исчисления предикатов. Таблица `Sales.Customers` с информацией о местонахождении клиентов состоит из трех атрибутов: `country`, `region` и `city`. Каждая запись содержит реальные названия стран и городов. Для большинства записей указан регион (`country`: США, `region`: WA, `city`: Сиэтл)<sup>1</sup>, но не для всех (`country`: Великобритания, `region`: NULL, `city`: Лондон). Рассмотрим запрос, который возвращает записи о клиентах, проживающих в регионе WA (штат Вашингтон).

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region = N'WA';
```

Результат будет следующим.

custid	country	region	city
43	США	WA	Уолла Уолла
82	США	WA	Киркланд
89	США	WA	Сиэтл

(строк обработано: 3)

Из 91 строки в таблице `Customers` запрос вернул три, в которых атрибут `region` равен «WA». Записи с другим или просто неопределенным регионом были отклонены (в первом случае предикат возвращает `FALSE`, а во втором `UNKNOWN`).

Следующий запрос пытается вернуть все строки с регионом, отличным от «WA».

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region <> N'WA';
```

Вот какой результат мы получим.

custid	country	region	city
10	Канада	BC	Тсаввассен
15	Бразилия	SP	Сан-Паулу
21	Бразилия	SP	Сан-Паулу
31	Бразилия	SP	Кампинас
32	США	OR	Юджин
33	Венесуэла	DF	Каракас
34	Бразилия	RJ	Рио-де-Жанейро
35	Венесуэла	Тачира	Сан-Кристобаль
36	США	OR	Элджин
37	Ирландия	Графство Корк	Корк
38	Великобритания	Остров Уайт	Коуз

<sup>1</sup> Названия регионов указаны в формате ISO 3166-2. — *Прим. переводчика.*



42	Канада	BC	Ванкувер
45	США	CA	Сан-Франциско
46	Венесуэла	Лара	Баркисимето
47	Венесуэла	Нуэва-Эспарта	Маргарита
48	США	OR	Портленд
51	Канада	Квебек	Монреаль
55	США	AK	Анкоридж
61	Бразилия	RJ	Рио-де-Жанейро
62	Бразилия	SP	Сан-Паулу
65	США	NM	Альбукерке
67	Бразилия	RJ	Рио-де-Жанейро
71	США	ID	Бойсе
75	США	WY	Ландер
77	США	OR	Портленд
78	США	MT	Бьют
81	Бразилия	SP	Сан-Паулу
88	Бразилия	SP	Резенди

(строк обработано: 28)

Возможно, вы ожидали получить не 28 строк, а 88 (91 минус три из предыдущего запроса). Однако фильтр «принимать TRUE» отклоняет строки, для которых логическое выражение возвращает FALSE или UNKNOWN. Результатом запроса стали записи, где атрибут region не равен «WA», то есть помимо других регионов проверку не прошли значения NULL. То же самое можно получить с помощью предиката NOT (region = N'WA'), поскольку неопределенное значение атрибута region приводит к результату UNKNOWN, а выражение NOT UNKNOWN также является неопределенным.

Если вы хотите получить строки, для которых регион равен NULL, не используйте предикат region = NULL, так как он всегда возвращает UNKNOWN, независимо от реального значения атрибута region. Следующий запрос вернет пустой набор.

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region = NULL;
```

custid	country	region	city
-----	-----	-----	-----

(строк обработано: 0)

В этой ситуации необходимо применить предикат IS NULL.

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region IS NULL;
```

Ниже представлена часть результата запроса.

custid	country	region	city
-----	-----	-----	-----
1	Германия	NULL	Берлин
2	Мексика	NULL	Мехико
3	Мексика	NULL	Мехико

4	Великобритания	NULL	Лондон
5	Швеция	NULL	Лулео
6	Германия	NULL	Мангейм
7	Франция	NULL	Страсбург
8	Испания	NULL	Мадрид
9	Франция	NULL	Марсель
11	Великобритания	NULL	Лондон
...			

(строка обработано: 60)

Если вы хотите получить записи, для которых атрибут `region` не равен «WA» (как с другими регионами, так и с неопределенными значениями), добавьте отдельную проверку на значение `NULL`.

```
SELECT custid, country, region, city
FROM Sales.Customers
WHERE region <> N'WA'
      OR region IS NULL;
```

Приведенным урезанный результат этого запроса.

custid	country	region	city
-----	-----	-----	-----
1	Германия	NULL	Верлин
2	Мексика	NULL	Мехико
3	Мексика	NULL	Мехико
4	Великобритания	NULL	Лондон
5	Швеция	NULL	Лулео
6	Германия	NULL	Мангейм
7	Франция	NULL	Страсбург
8	Испания	NULL	Мадрид
9	Франция	NULL	Марсель
10	Канада	BC	Тсаввассен

(строка обработано: 88)

Различные элементы языка `SQL` по-разному обрабатывают отметки `NULL` при сравнении и сортировке. Иногда два значения `NULL` могут быть равными, а иногда нет.

Например, при группировании и сортировке отметки `NULL` считаются одинаковыми, то есть инструкция `GROUP BY` поместит все пустые значения в одну группу. Аналогичные действия произойдут при использовании инструкции `ORDER BY`. Спецификация языка `SQL` не содержит строгих правил насчет того, где должны размещаться отметки `NULL` — до или после определенных значений. В `T-SQL` используется второй вариант.

Как упоминалось ранее, фильтры «принимают `TRUE`». Выражение, которое сравнивает две отметки `NULL`, возвращает `UNKNOWN`; следовательно, такие строки отсеиваются.

Чтобы ограничения `UNIQUE` соблюдались, `SQL` относится к отметкам `NULL` как к разным значениям (позволяя иметь множество неопределенных отметок). Однако в `T-SQL` все наоборот: в рамках ограничения `UNIQUE` два значения `NULL` считаются равными (если ограничение определено только для одного столбца, допускается лишь одна отметка `NULL`).

Помните, что в языке SQL обработка значений UNKNOWN и NULL зависит от контекста и может приводить к логическим ошибкам. К каждому запросу подходит индивидуально, обращая внимание на отметки NULL и троичную логику. Сами определяйте условия, которые вас устраивают; в противном случае убедитесь, что вам действительно подходит стандартное поведение.

## Одновременные операции

Язык SQL поддерживает концепцию **одновременных операций**. Это значит, что все выражения, которые находятся на одном этапе обработки, логически выполняются одновременно.

Именно поэтому в инструкции SELECT нельзя ссылаться на псевдонимы столбцов, присвоенные в том же списке. Рассмотрим следующий запрос.

```
SELECT
    orderid,
    YEAR(orderdate) AS orderyear,
    orderyear + 1 AS nextyear
FROM Sales.Orders;
```

Ссылка на псевдоним в третьем выражении внутри инструкции SELECT является некорректной, несмотря на то что она указана после назначения псевдонима, поэтому запрос генерирует ошибку.

```
Сообщение 207, уровень 16, состояние 1, строка 4
Недопустимое имя столбца "orderyear".
```

Рассмотрим пример, который демонстрирует важность одновременных операций. Допустим, таблица T1 содержит целочисленные столбцы col1 и col2, вы хотите получить все строки, для которых выражение col2/col1 больше 2. Поскольку в столбце col1 могут храниться нули, вы должны убедиться, что такие значения не попадут в знаменатель, иначе появится ошибка деления на ноль. Итак, вы написали запрос.

```
SELECT col1, col2
FROM dbo.T1
WHERE col1 <> 0 AND col2/col1 > 2;
```

Логично предположить, что SQL Server обрабатывает выражения слева направо, и если col1 <> 0 вернет FALSE, второе условие не будет выполнено. Другими словами, нет смысла вызывать 10/col1, если известно, что в целом выражение возвращает FALSE. Поэтому можно надеяться, что данный запрос никогда не сгенерирует ошибку деления на ноль.

SQL Server поддерживает сокращенные вычисления, но из-за концепции одновременного выполнения выражения в инструкции WHERE могут обрабатываться в любом порядке. Обычно программа исходит из потенциальной производительности, то есть первым выполняется выражение, которое требует меньше всего системных ресурсов. Таким образом, если SQL Server решит, что сначала нужно обработать операцию 10/col1 > 2, запрос завершится неудачно.

Этих неприятностей можно избежать несколькими способами. Например, выражение CASE всегда выполняет вложенные инструкции WHEN последовательно, поэтому перепишите запрос следующим образом.

```
SELECT col1, col2
FROM dbo.T1
WHERE
  CASE
    WHEN col1 = 0 THEN 'нет' -- или 'да', если строка должна быть
    возвращена
    WHEN col2/col1 > 2 THEN 'да'
    ELSE 'нет'
  END = 'да';
```

Для строки, в которой атрибут col1 равен 0, первая инструкция WHEN вернет TRUE, а результатом выполнения выражения CASE будет строка 'нет' (замените 'нет' на 'да', если хотите получить строки, где col1 равен нулю). Только в том случае, когда первая инструкция в выражении CASE возвращает не TRUE (если col1 не равно 0), запускается вторая. В ней на истинность проверяется предикат col2/col1 > 2. Если ответ положителен, выражение CASE возвращает 'да'. В остальных случаях мы получим строку 'нет'. Предикат в главной инструкции WHERE является истинным, лишь когда выражение CASE равно строке 'да'. Здесь мы никогда не столкнемся с попыткой деления на нуль.

Приведенное выше решение довольно запутанно. Конкретно в данном случае вы можете воспользоваться математическим приемом, чтобы изначально исключить деление на нуль.

```
SELECT col1, col2
FROM dbo.T1
WHERE (col1 > 0 AND col2 > 2*col1) OR (col1 < 0 AND col2 < 2*col1);
```

Я добавил этот пример, чтобы подчеркнуть важность концепции одновременных операций и обратить внимание на то, что SQL Server соблюдает порядок выполнения инструкций WHEN внутри выражения CASE.

## Работа с символьными данными

В этом разделе мы обсудим обработку символьной информации внутри запросов. Я отдельно остановлюсь на типах данных, операциях сравнения, операторах, функциях и сопоставлении по шаблону.

### Типы данных

SQL Server поддерживает два вида символьных данных: обычные и представленные в кодировке Unicode. К первым относятся типы CHAR и VARCHAR, ко вторым — NCHAR и NVARCHAR. Для хранения обычного символа достаточно одного байта; в Unicode символы имеют разный размер (обычно два байта, но в случае с суррогатными парами может достигать четырех). Выбрав для столбца обычный тип, вы будете ограничены двумя языками, один из которых английский. Поддержка языков определяется правилами сравнения, которые действуют

в столбце (об этом поговорим позже). Типы данных на основе кодировки Unicode поддерживают множество языков. Поэтому, если вы собираетесь хранить символы из разных алфавитов, используйте типы `NCHAR` или `NVARCHAR`.

Для записи строки, состоящей из обычных символов, используются одинарные кавычки: `'Это обычная строка'`. При использовании кодировки Unicode необходимо указать префикс `N`: `N'Это строка в формате Unicode'`.

Любой тип данных, в названии которого нет приставки `VAR` (`CHAR`, `NCHAR`), имеет фиксированную длину. Это значит, что `SQL Server` выделяет место в строке, учитывая заранее определенный размер столбца, а не реальное количество символов. Например, если тип столбца `CHAR (25)`, ему выделяется 25 символов, и не важно, строку какой длины он хранит. Такие типы больше подходят для систем, ориентированных на запись данных, поскольку строки таблицы не должны расширяться вместе с содержимым. В условиях когда основной операцией является чтение, появляется проблема неоптимального использования дискового пространства.

Типы данных с приставкой `VAR` в названии (`VARCHAR`, `NVARCHAR`) обладают переменной длиной: им выделяется столько места, сколько занимают хранимые в них строки (плюс два дополнительных байта для разделения столбцов). Допустим, столбец типа `VARCHAR (25)` хранит не более 25 символов, но на самом деле его размер определяется объемом содержимого. Поскольку эти типы данных потребляют меньше дискового пространства, они обеспечивают лучшую производительность при чтении. Однако во время обновления строки таблицы могут расширяться. В итоге данные выходят за пределы текущей страницы и, как следствие, замедляются операции записи.



#### ПРИМЕЧАНИЕ

Использование сжатия влияет на требования к дисковому пространству. Подробнее об этом читайте в разделе «Сжатие данных» электронного справочника по адресу [msdn.microsoft.com/ru-ru/library/cc280449.aspx](https://msdn.microsoft.com/ru-ru/library/cc280449.aspx).

Типы данных переменной длины поддерживают спецификатор `MAX`. Он определяет максимальный объем содержимого не в символах, а в байтах (8000 байт по умолчанию). Это значение касается не отдельного столбца, а всей строки. Все, что выходит за рамки определенного размера, сохраняется в дополнительной строке в виде больших объектов (`large object`, или `LOB`).

В разделе «Извлечение метаданных» я объясню, как получить метainформацию об объектах (включая типы данных столбцов), хранящихся в БД.

## Параметры сравнения

Операция сравнения символьных данных включает поддержку языков, порядок сортировки, чувствительность к регистру и диакритическим знакам. Чтобы получить список поддерживаемых параметров и их описание, можно выполнить запрос к табличной функции `fn_helpcollations`.

```
SELECT name, description
FROM sys.fn_helpcollations();
```

В следующем списке рассматривается параметр `Latin1_General_CI_AS`.

- **Latin1\_General.** Применяется кодовая страница 1252 (поддерживает символы английского и немецкого алфавитов, распространенных в большинстве западноевропейских стран).
- **Сортировка.** Символы сортируются и сравниваются на основе алфавитного порядка («А» и «а» < «В» и «b»).
- **Алфавитный порядок.** Используется по умолчанию. Если в названии параметра содержится элемент `BIN`, сортировка и сравнение данных будут выполняться с учетом двоичного представления символов («А» < «В» < «а» < «b»).
- **CI.** Данные не чувствительны к регистру («а» = «А»).
- **AS.** Учитываются диакритические знаки («à» <> «ä»).

В локальной версии SQL Server параметры сравнения можно определить на уровне экземпляра, БД, столбца и выражения. На практике лучше это делать на самом низком уровне. В Windows Azure SQL Database вам доступны все уровни, кроме экземпляра.

Параметры сравнения для всего экземпляра выбираются на этапе установки. Они относятся ко всем системным БД и применяются по умолчанию в пользовательских БД.

Создавая БД, вы можете переопределить эти параметры с помощью инструкции `COLLATE`.

Параметры, определенные на уровне БД, касаются метаданных об объектах и используются по умолчанию в столбцах пользовательских таблиц. Хочу подчеркнуть, что эти параметры описывают правила сравнения метаданных, включая имена объектов и столбцов. Например, если параметр чувствителен к регистру, в рамках одной схемы существует возможность создать две таблицы с именами `T1` и `t1`.

Параметр сравнения можно указать вручную в определении каждого столбца, используя инструкцию `COLLATE`. С ее помощью переопределяются ранее заданные параметры. Например, мы можем отключить чувствительность к регистру в своем запросе.

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname = N'дэвис';
```

В итоге мы получим запись о Саре Дэвис, хотя ее фамилия была записана с маленькой буквы.

```
empid      firstname  lastname
-----
1          Сара      Дэвис

(строк обработано: 1)
```

Если вы хотите обеспечить чувствительность к регистру там, где она не поддерживается, измените параметры сравнения в самом выражении.

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname COLLATE Latin1_General_CS_AS = N'дэвис';
```

На этот раз запрос вернет пустой набор, поскольку сравнение с учетом регистра не даст ни единого совпадения.



### КАВЫЧКИ

В стандартной разновидности языка SQL в одинарные кавычки заключаются символьные строки ( ' строка ' ), а в двойные — различные идентификаторы, например имена таблиц или столбцов, которые содержат пробел или начинаются с цифры ( " специальный идентификатор " ). В SQL Server существует параметр `QUOTED_IDENTIFIER`. Он определяет значение двойных кавычек. Параметр применяется как на уровне БД, внутри команды `ALTER DATABASE`, так и в рамках одной сессии с использованием команды `SET`. В активированном состоянии этот параметр обеспечивает соответствие стандарту. Когда он выключен, двойные кавычки можно использовать и в символьных строках. Желательно, чтобы вы применяли стандартный вариант с активированным параметром. В большинстве интерфейсов БД, таких как OLEDB и ODBC, он включен по умолчанию.

Если вы хотите ввести символ одиночной кавычки внутри строки, укажите его два раза подряд. Например, строка «абв'гд» будет выглядеть как ' абв ' ' гд ' .



### ПОДСКАЗКА

В SQL Server выделять идентификаторы можно не только двойными кавычками, но и с помощью квадратных скобок — [специальный идентификатор].

## Операторы и функции

В этом разделе мы обсудим функции, необходимые для работы со строками, и отдельно остановимся на операции объединения. В языке T-SQL объединять строки можно с помощью оператора `+` и функции `CONCAT`. Для других операций со строками предусмотрены функции `SUBSTRING`, `LEFT`, `RIGHT`, `LEN`, `DATALength`, `CHARINDEX`, `PATINDEX`, `REPLACE`, `REPLICATE`, `STUFF`, `UPPER`, `LOWER`, `RTRIM`, `LTRIM` и `FORMAT`. В следующих подразделах я опишу самые популярные из них.

### Объединение строк (оператор `+` и функция `CONCAT`)

Для объединения строк в языке T-SQL существует два инструмента: оператор `+` (знак «плюс») и функция `CONCAT` (начиная с SQL Server 2012). В результате следующего запроса к таблице `Employees` мы получим столбец `fullname`. В нем присутствуют значения атрибутов `firstname` и `lastname`, разделенные пробелом.

```
SELECT empid, firstname + N' ' + lastname AS fullname
FROM HR.Employees;
```

Вот какой результат вернет запрос.

empid	fullname
1	Сара Дэвис

```
2      Дон Функ
3      Джуди Лью
4      Иаиль Пелед
5      Свен Бак
6      Пол Суурс
7      Рассел Кинг
8      Мария Камерон
9      Зоя Долгопятова
```

(строк обработано: 9)

В стандартной разновидности языка SQL объединение со значением NULL возвращает NULL. SQL Server соблюдает это правило по умолчанию. Возьмем для примера запрос к таблице **Customers**, представленный в листинге 2.7.

**Листинг 2.7.** Запрос, демонстрирующий объединение строк

```
SELECT custid, country, region, city,
       country + N', ' + region + N', ' + city AS location
FROM Sales.Customers;
```

Некоторые строки в таблице **Customers** в качестве региона содержат пустое значение. Для них в столбце **location** по умолчанию возвращается NULL.

custid	country	region	city	location
1	Германия	NULL	Берлин	NULL
2	Мексика	NULL	Мехико	NULL
3	Мексика	NULL	Мехико	NULL
4	Великобритания	NULL	Лондон	NULL
5	Швеция	NULL	Лулео	NULL
6	Германия	NULL	Мангейм	NULL
7	Франция	NULL	Страсбург	NULL
8	Испания	NULL	Мадрид	NULL
9	Франция	NULL	Марсель	NULL
10	Канада	BC	Тсаввассен	Канада, BC, Тсаввассен
11	Великобритания	NULL	Лондон	NULL
12	Аргентина	NULL	Буэнос-Айрес	NULL
13	Мексика	NULL	Мехико	NULL
14	Швейцария	NULL	Берн	NULL
15	Бразилия	SP	Сан-Паулу	Бразилия, SP, Сан-Паулу
16	Великобритания	NULL	Лондон	NULL
17	Германия	NULL	Ахен	NULL
18	Франция	NULL	Нант	NULL
19	Великобритания	NULL	Лондон	NULL
20	Австрия	NULL	Грац	NULL

...  
(строк обработано: 91)

Чтобы заменить значение NULL пустой строкой, воспользуйтесь функцией COALESCE. Она принимает список входящих аргументов и возвращает первый из них, который не равен NULL. Перепишем запрос из листинга 2.7, чтобы вместо отметок NULL автоматически вставлялись пустые строки.

```
SELECT custid, country, region, city,
       country + COALESCE( N', ' + region, N'' ) + N', ' + city AS location
FROM Sales.Customers;
```



В SQL Server 2012 появилась новая функция под названием `CONCAT`. Она принимает список аргументов для их последующего объединения; замена отметок `NULL` пустыми строками здесь происходит автоматически. Например, выражение `CONCAT('a', NULL, 'b')` возвращает строку `'ab'`.

Вот как с помощью функции `CONCAT` можно объединить элементы местоположения клиента, вставляя вместо значений `NULL` пустые строки.

```
SELECT custid, country, region, city,  
       CONCAT(country, N', ' + region, N', ' + city) AS location  
FROM Sales.Customers;
```

## Функция `SUBSTRING`

Возвращает определенный фрагмент заданной строки.

### Синтаксис:

`SUBSTRING(строка, начало, длина)`

Эта функция принимает входящую строку и извлекает из нее фрагмент заданной длины, начиная с определенной позиции. Следующий код возвращает `'abc'`:

```
SELECT SUBSTRING('abcde', 1, 3);
```

Если значение третьего аргумента выйдет за пределы исходной строки, функция ограничит результат последним символом, не выдавая ошибки. Это удобно, когда требуется фрагмент начиная с определенной позиции и до конца — вы просто указываете максимальную длину типа данных или значение, представляющее полную длину входящей строки.

## Функции `LEFT` и `RIGHT`

Являются сокращенными вариантами `SUBSTRING`. Они возвращают заданное количество символов, начиная с левого или правого края строки.

### Синтаксис:

`LEFT(строка, n)`, `RIGHT(строка, n)`

В качестве первого аргумента выступает строка, с которой будут проводиться манипуляции. Второй аргумент, `n`, указывает количество символов, которые необходимо извлечь справа или слева. Следующий код возвращает подстроку `'cde'`.

```
SELECT RIGHT('abcde', 3);
```

## Функции `LEN` и `DATALength`

Возвращает количество символов в заданной строке.

### Синтаксис:

`LEN(строка)`

Количество символов в строке не всегда совпадает с количеством байтов. В обычных строках каждый символ равен одному байту; в кодировке `Unicode`

символы занимают в основном два байта. Таким образом, полученный результат может быть в два раза меньше, чем размер строки. Для измерения строк в байтах существует функция `DATALength`. Например, следующий код вернет 5.

```
SELECT LEN(N'abcde');
```

А результатом этого выражения будет 10.

```
SELECT DATALength(N'abcde');
```

Кроме того, в отличие от `DATALength` функция `LEN` вырезает замыкающие пробельные символы.

## Функция `CHARINDEX`

Возвращает позицию первого вхождения одной строки в другую.

### Синтаксис:

```
CHARINDEX(подстрока, строка[, нач_поз])
```

Функция возвращает местоположение первого аргумента (подстроки) во втором (строке). Вы также можете указать третий аргумент, чтобы сообщить позицию, с которой следует искать вхождения (по умолчанию поиск начинается с первого символа). Если подстрока не найдена, функция возвращает 0. Приведенный ниже код позволяет получить позицию первого пробела в строке 'Ицик Бен-Ган', то есть число 5.

```
SELECT CHARINDEX(' ', 'Ицик Бен-Ган');
```

## Функция `PATINDEX`

Возвращает позицию первого вхождения шаблона в строку.

### Синтаксис:

```
PATINDEX(шаблон, строка)
```

В первом аргументе используются те же шаблоны, что и в предикате `LIKE` (о нем поговорим в разделе «Предикат `LIKE`» текущей главы). И хотя мы не дошли до работы с шаблонами в языке T-SQL, я приведу небольшой пример. Так мы найдем позицию первого вхождения в строку любой цифры.

```
SELECT PATINDEX('%[0-9]%', 'abcd123efgh');
```

Код вернет число 5.

## Функция `REPLACE`

Заменяет все вхождения одной строки другой строкой.

### Синтаксис:

```
REPLACE(строка, подстрока1, подстрока2)
```

Все вхождения подстроки1 в строку заменяются подстрокой2. Например, следующий код подставляет вместо дефисов двоеточия.

```
SELECT REPLACE('1-a 2-b', '-', ':');
```

Результатом будет строка '1:a 2:b'.

С помощью функции REPLACE можно подсчитывать количество вхождений символа в строку. Для этого каждое вхождение нужно заменить пустой строкой, а затем вычесть длину результата из длины исходной строки. Приведенный ниже запрос поможет узнать, сколько букв «o» содержится в фамилии каждого сотрудника.

```
SELECT empid, lastname,
       LEN(lastname) - LEN(REPLACE(lastname, 'o', '')) AS numoccur
FROM HR.Employees;
```

Вот какой результат мы получим.

empid	lastname	numoccur
5	Бак	0
9	Долгопятова	3
1	Дэвис	0
8	Камерон	1
7	Кинг	0
3	Лью	0
4	Пелед	0
6	Суурс	0
2	Функ	0

(строк обработано: 9)

## Функция REPLICATE

Повторяет значение строки заданное количество раз.

### Синтаксис:

```
REPLICATE(строка, n)
```

Код, представленный ниже, повторяет строку 'abc' три раза и возвращает значение 'abcabcabc'.

```
SELECT REPLICATE('abc', 3);
```

В следующем примере функция REPLICATE используется вместе с операцией объединения строк и функцией RIGHT. Наш запрос обращается к таблице **Production.Suppliers** и выводит идентификаторы поставщиков в десятизначном представлении, добавляя в начало нули.

```
SELECT supplierid,
       RIGHT(REPLICATE('0', 9) + CAST(supplierid AS VARCHAR(10)), 10) AS
       strsupplierid
FROM Production.Suppliers;
```

Это выражение создает столбец под названием `strsupplierid`, повторяя значение 0 девять раз (что в итоге дает строку '000000000'), и объединяет результат со строкой, представляющей идентификатор поставщика. Чтобы превратить целочисленный идентификатор в строку, используется функция `CAST`, которая конвертирует тип данных заданного значения. В конце выражение извлекает десять последних символов и возвращает десятизначное строковое представление идентификатора поставщика с нулями в начале. Ниже представлен урезанный результат данного запроса.

supplierid	strsupplierid
-----	-----
29	0000000029
28	0000000028
4	0000000004
21	0000000021
2	0000000002
22	0000000022
14	0000000014
11	0000000011
25	0000000025
7	0000000007
...	

(строка обработано: 29)

В SQL Server 2012 появилась новая функция под названием `FORMAT`. Она позволяет добиться подобного форматирования более простым путем. Мы рассмотрим ее позже в этом разделе.

## Функция STUFF

Удаляет из строки заданный фрагмент и вставляет на его место новый.

### Синтаксис:

`STUFF(строка, позиция, длина_фрагмента, новый_фрагмент)`

Эта функция оперирует исходной строкой. Она удаляет последовательность символов определенной длины, начиная с указанной позиции. Затем на то же место помещает новый фрагмент. Следующий код удаляет из строки 'xyz' один символ под номером 2 и добавляет вместо него строку 'abc'.

```
SELECT STUFF('xyz', 2, 1, 'abc');
```

Результатом будет значение 'xabcz'.

Если вы хотите вставить строку, ничего не удаляя, в качестве третьего аргумента укажите 0.

## Функции UPPER и LOWER

Переводят заданную строку в верхний или нижний регистр соответственно.

### Синтаксис:

`UPPER(строка), LOWER(строка)`

Результатом выполнения следующего кода будет строка 'ИЦИК БЕН-ГАН'.

```
SELECT UPPER('Ицик Бен-Ган');
```

Запрос, представленный ниже, возвращает значение 'ицик бен-ган'.

```
SELECT LOWER('Ицик Бен-Ган');
```

## Функции RTRIM и LTRIM

Возвращают исходную строку, удаляя из нее начальные или завершающие пробелы соответственно.

### Синтаксис:

```
RTRIM(строка) , LTRIM(строка)
```

Если вы хотите удалить и начальные, и завершающие пробелы, используйте результат выполнения одной функции в качестве аргумента для другой. К примеру, следующий код вернет значение 'abc'.

```
SELECT RTRIM(LTRIM(' abc '));
```

## Функция FORMAT

Преобразовывает заданное значение, используя региональные параметры и формат, принятый на платформе Microsoft .NET.

### Синтаксис:

```
FORMAT(ввод, формат, региональные_параметры)
```

Существует множество вариантов, как преобразовать исходное значение; для этого можно использовать как стандартные, так и собственные строки форматирования. Больше информации на эту тему вы найдете в статье по адресу [go.microsoft.com/fwlink/?LinkId=211776](https://go.microsoft.com/fwlink/?LinkId=211776).

Возьмем для примера выражение, с помощью которого мы представляли число в виде десятизначной строки с нулями в начале. Функция FORMAT позволяет достичь такого же результата — достаточно указать строку форматирования '000000000' (или ее стандартный аналог 'd10'). Следующий код вернет значение '0000001759'.

```
SELECT FORMAT(1759, '000000000');
```

## Предикат LIKE

В языке T-SQL существует предикат под названием LIKE. Он проверяет, соответствует ли строка заданному шаблону. Похожие шаблоны используются в функции PATINDEX, которую мы рассматривали выше. Далее я расскажу, какие групповые символы поддерживаются в T-SQL и как их использовать.

## Групповой символ % (процент)

Знак процента представляет строку произвольного (в том числе нулевого) размера. Следующий запрос возвращает сотрудников, чьи фамилии начинаются на букву «Д».

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'Д%';
```

Вот какой результат мы получим.

empid	lastname
9	Долгопятова
1	Дэвис

Во многих случаях предикат LIKE можно заменить функциями SUBSTRING и LEFT, однако помните, что он лучше оптимизирован (особенно если шаблон начинается с известного префикса).

## Групповой символ \_ (подчеркивание)

Знак подчеркивания представляет одиночный символ. Приведенный ниже запрос возвращает сотрудников, в фамилии которых буква «у» стоит на втором месте.

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'_у%';
```

Результат представлен ниже.

empid	lastname
6	Суурс
2	Функ

## Групповые символы [<список\_символов>]

Квадратные скобки используются для определения списка символов (например, [ABC]), один из которых должен соответствовать шаблону. Следующий запрос возвращает сотрудников, фамилии которых начинаются с букв «А», «Б» или «Л».

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'[АБЛ]%';
```

Результат будет таким.

empid	lastname
5	Бак
3	Лью

## Групповые символы [<символ>-<символ>]

В квадратных скобках можно задать диапазон символов (например, [A-E]), один из которых должен соответствовать шаблону. Запрос, представленный ниже, возвращает сотрудников, фамилии которых начинаются с букв, лежащих в диапазоне от «А» до «К».

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'[A-K]%' ;
```

Вот какой результат мы получим.

empid	lastname
5	Бак
9	Долгопятова
1	Дэвис
8	Камерон
7	Кинг

## Групповые символы [^<список или диапазон символов>]

Символ вставки (^), расположенный перед списком или диапазоном символов внутри квадратных скобок (например, [^A-E]), играет роль логического отрицания. Шаблону будет соответствовать любой одиночный символ, не входящий в список или диапазон. Ниже показан запрос, который возвращает сотрудников, чьи фамилии не начинаются с букв от «А» до «К».

```
SELECT empid, lastname
FROM HR.Employees
WHERE lastname LIKE N'^[A-K]%' ;
```

Мы получим следующий результат.

empid	lastname
3	Лью
4	Пелед
6	Суурс
2	Функ

## Символ экранирования

Если вы хотите найти символ, который по совпадению является групповым (% , \_ , [ или ]), можете его экранировать. Для этого используйте символ экранирования. Его необходимо указать перед искомым групповым символом и после ключевого слова ESCAPE, которое следует за шаблоном. Например, чтобы узнать, содержит ли столбец col1 символ подчеркивания, воспользуемся следующим выражением: col1 LIKE '%!\_%' ESCAPE '!'.

Для групповых символов %, \_ и [ ] вместо экранирования также применяются квадратные скобки. Таким образом, предыдущее выражение можно записать как col1 LIKE '%[\_]%'.

# Работа с датой и временем

Работа с датой и временем в SQL Server — довольно сложный процесс. Вас ждут определенные трудности, связанные с разделением даты и времени, а также со стилем описания строковых литералов, который не конфликтует с языком SQL.

Первым делом мы познакомимся с соответствующими типами данных, которые поддерживаются в SQL Server; затем я расскажу, как лучше подходить к работе с ними; в конце мы рассмотрим функции, относящиеся к дате и времени.

## Типы данных даты и времени

До выхода SQL Server 2008 язык T-SQL поддерживал всего два типа данных, связанных с датой и временем, — DATETIME и SMALLDATETIME. Они являются монолитными (не позволяя отделить дату от времени) и отличаются друг от друга количеством занимаемого места, диапазоном поддерживаемых дат и точностью значений. В SQL Server появились отдельные типы, DATE и TIME, а также типы DATETIME2 (с расширенным диапазоном и улучшенной точностью) и DATETIMEOFFSET (с поддержкой часовых поясов). Описание типов данных даты и времени приведено в таблице 2.1.

Таблица 2.1. Типы даты и времени

Тип данных	Объем	Диапазон	Точность	Рекомендуемый формат записей и пример
DATETIME	8	с 1 января 1753 по 31 декабря 9999	3,333 миллисекунды	'YYYYMMDD hh:mm:ss.nnn' '20090212 12:30:15.123'
SMALLDATETIME	4	с 1 января 1900 по 6 июня 2079	1 минута	'YYYYMMDD hh:mm' '20090212 12:30'
DATE	3	с 1 января 0001 по 31 декабря 9999	1 день	'YYYY-MM-DD' '2009-02-12'
TIME	от 3 до 5	—	100 наносекунд	'hh:mm:ss.nnnnnnn' '12:30:15.1234567'
DATETIME2	от 6 до 8	с 1 января 0001 по 31 декабря 9999	100 наносекунд	'YYYY-MM-DD hh:mm:ss.nnnnnnn' '2009-02-12 12:30:15.1234567'
DATETIMEOFFSET	от 8 до 10	с 1 января 0001 по 31 декабря 9999	100 наносекунд	'YYYY-MM-DD hh:mm:ss.nnnnnnn [+ -]hh:mm' '2009-02-12 12:30:15.1234567 +02:00'

Объем, который занимают типы данных TIME, DATETIME2 и DATETIMEOFFSET, зависит от выбранной точности. Точность указывается в виде целого числа от 0 до 7 и представляет дробную часть секунды. Например, TIME(0) означает, что время отсчитывается по одной секунде, TIME(3) — по одной миллисекунде, TIME(7) — по 100 наносекунд. Во всех трех типах по умолчанию используется наивысший, седьмой, уровень точности.



## Строковые литералы

Представляя дату и время в виде строковых литералов (констант), необходимо учитывать несколько моментов. SQL Server не содержит средств, позволяющих описывать дату и время в виде строк. Вы должны сперва создать литерал другого типа, а затем преобразовать его автоматически или вручную. Рекомендую использовать для этого символьные строки, как показано в следующем примере.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20070212';
```

SQL Server воспринимает литерал '20070212' как строковую константу, а не как обозначение даты и времени. Однако в выражении участвуют операнды двух разных типов, поэтому один из них должен автоматически трансформироваться в другой. Обычно такое неявное преобразование выполняется с учетом приоритетов типов данных, определенных на уровне SQL Server: операнд автоматически приводится к более приоритетному типу.

В примере строковый литерал преобразуется в тип данных столбца (DATETIME), потому что строка менее приоритетна, чем дата и время. Правила неявного приведения типов не всегда работают так просто — например, свои особенности есть у фильтров и других выражений. Полное описание приоритетов типов данных вы найдете в электронном справочнике в разделе «Приоритет типов данных (Transact-SQL)».

Итак, неявное приведение типов выполняется автоматически. А теперь превратим строку в значение типа DATETIME вручную.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = CAST('20070212' AS DATETIME);
```

Некоторые строковые представления даты и времени зависят от языковых настроек SQL Server и могут интерпретироваться по-разному. С каждой сессией, созданной администратором БД, связана стандартная конфигурация, которая определяет используемый язык. У вас есть возможность переопределить языковые параметры сессии с помощью команды SET LANGUAGE. Однако учтите, что код некоторых запросов может быть рассчитан только на определенный язык, применяемый по умолчанию.

Помимо самого языка для сессии автоматически устанавливается несколько сопутствующих параметров. Один из них называется DATEFORMAT. Он определяет то, как SQL Server интерпретирует строковые литералы, которые преобразовываются в значения даты и времени. Параметр DATEFORMAT задается в виде трех символов: d, m и y. Например, в языке us\_english используется формат mdy, а в языке Russian — dmy. Параметр DATEFORMAT можно переопределить для текущей сессии, используя команду SET DATEFORMAT, но, как было сказано выше, изменять языковые настройки в целом не рекомендуется.

Рассмотрим строковый литерал '02/12/2007'. Если вы попытаетесь привести его к типам DATETIME, DATE, DATETIME2 или DATETIMEOFFSET, SQL Server

истолкует дату либо как 12 февраля 2007 г., либо как 2 декабря 2007 г. Основным фактором являются настройки `LANGUAGE/DATEFORMAT`. Чтобы увидеть разницу в трактовке одного и того же литерала, запустите следующий код.

```
SET LANGUAGE Russian;
SELECT CAST('02/12/2007' AS DATETIME);
```

```
SET LANGUAGE us_english;
SELECT CAST('02/12/2007' AS DATETIME);
```

Итоговый результат зависит от текущей языковой среды.

```
Параметры языка изменены на "русский".
```

```
-----
2007-12-02 00:00:00.000
```

```
Changed language setting to us_english.
```

```
-----
2007-02-12 00:00:00.000
```

Параметры `LANGUAGE/DATEFORMAT` влияют только на интерпретацию вводимых значений. Они не относятся к формату представления даты и времени, который определяется в интерфейсе БД с помощью клиентских инструментов (таких как ODBC). Например, OLEDB и ODBC выводят значения типа `DATETIME` в формате `'YYYY-MM-DD hh:mm:ss.nnn'`.

Код, который вы напишете, могут использовать люди из разных стран с разнообразными языковыми настройками, поэтому важно понимать, что формат строковых литералов зависит от языка. Я рекомендую представлять литералы в нейтральном виде, тогда они будут толковаться одинаково и на них не повлияют различия в региональных стандартах. В таблице 2.2 перечислены нейтральные форматы литералов для каждого типа даты и времени.

Обратите внимание на несколько моментов. Если в составном типе (который хранит дату и время) не указать значение времени, SQL Server автоматически поставит «полночь». Если опустить часовой пояс, будет предполагаться, что он равен 00:00. Форматы `'YYYY-MM-DD'` и `'YYYY-MM-DD hh:mm. . .'` являются нейтральными для типов `DATETIME` и `SMALLDATETIME`, но зависят от языка, если литерал преобразовывается в типы `DATE`, `DATETIME2` или `DATETIMEOFFSET`.

В следующем коде языковые настройки не влияют на интерпретацию литерала вида `'YYYYMMDD'`, когда он приводится к типу `DATETIME`.

```
SET LANGUAGE Russian;
SELECT CAST('20070212' AS DATETIME);
```

```
SET LANGUAGE us_english;
SELECT CAST('20070212' AS DATETIME);
```

Судя по результату, представленному ниже, литерал в любом случае превращается в дату «12 февраля 2007».

```
Параметры языка изменены на "русский".
-----
2007-02-12 00:00:00.000

Changed language setting to us_english.

-----
2007-02-12 00:00:00.000
```

Таблица 2.2. Нейтральные форматы представления даты и времени

Тип данных	Точность	Рекомендуемый формат записи и пример
DATETIME	'YYYYMMDD hh:mm:ss.nnn' 'YYYY-MM-DDThh:mm:ss.nnn' 'YYYYMMDD'	'20090212 12:30:15.123' '2009-02-12T12:30:15.123' '20090212'
SMALLDATETIME	'YYYYMMDD hh:mm' 'YYYY-MM-DDThh:mm' 'YYYYMMDD'	'20090212 12:30' '2009-02-12T12:30' '20090212'
DATE	'YYYYMMDD' 'YYYY-MM-DD'	'20090212' '2009-02-12'
DATETIME2	'YYYYMMDD hh:mm:ss.nnnnnnn' 'YYYY-MM-DD hh:mm:ss.nnnnnnn' 'YYYY-MM-DDThh:mm:ss.nnnnnnn' 'YYYYMMDD' 'YYYY-MM-DD'	'20090212 12:30:15.1234567' '2009-02-12 12:30:15.1234567' '2009-02-12T12:30:15.1234567' '20090212' '2009-02-12'
DATETIMEOFFSET	'YYYYMMDD hh:mm:ss.nnnnnnn [+ -]hh:mm' 'YYYY-MM-DD hh:mm:ss.nnnnnnn [+ -]hh:mm' 'YYYYMMDD' 'YYYY-MM-DD'	'20090212 12:30:15.1234567 +02:00' '2009-02-12 12:30:15.1234567 +02:00' '20090212' '2009-02-12'
DATETIMEOFFSET	'hh:mm:ss.nnnnnnn'	'12:30:15.1234567'

Если вы хотите следовать формату, принятому в определенном языке, существует два варианта. Первый — конвертировать строковые литералы в нужные типы данных с помощью функции CONVERT. При этом в качестве ее третьего аргумента указывается число, представляющее выбранный стиль. В электронном справочнике по SQL Server в разделе «Функции CAST и CONVERT (Transact-SQL)» приведена таблица со всеми значениями и описанием форматов, которые они представляют. Например, чтобы указать литерал '02/12/2007' в формате mm/dd/yyyy, используйте число 101.

```
SELECT CONVERT(DATETIME, '02/12/2007', 101);
```

Литерал будет интерпретирован как 12 февраля 2007 г. независимо от действующих языковых настроек.

Если необходимо использовать формат `dd/mm/yyyy`, укажите стиль 103.

```
SELECT CONVERT(DATETIME, '02/12/2007', 103);
```

На этот раз мы получим 2 декабря 2007 г.

Еще один вариант — использовать функцию `PARSE`, которая впервые появилась в SQL Server 2012. Она интерпретирует значение с учетом заданного типа и региональных особенностей. Например, ниже представлен запрос, эквивалентный выполнению функции `CONVERT` со стилем 101 (`us_english`).

```
SELECT PARSE('02/12/2007' AS DATETIME USING 'en-US');
```

Следующим кодом можно заменить функцию `CONVERT` со стилем 103 (`British`).

```
SELECT PARSE('02/12/2007' AS DATETIME USING 'en-GB');
```

## Раздельная работа с датой и временем

Только в SQL Server 2008 появились типы данных `DATE` и `TIME`, которые разделили работу с датой и временем. В предыдущих версиях были доступны лишь типы `DATETIME` и `SMALLDATETIME`, содержащие оба компонента. Раньше работу с датой и временем осуществляли на основе целочисленного или строкового типа, но мы не будем рассматривать этот вариант. Если вы хотите использовать типы `DATETIME` и `SMALLDATETIME` и при этом вам нужна только дата, укажите везде полночь (все нули в той части, которая представляет время). Чтобы работать исключительно со временем, используйте начальную дату — 1 января 1900 г.

Например, столбец `orderdate` в таблице `Sales.Orders` имеет тип `DATETIME`, но нас интересуют только даты, поэтому все значения «были записаны» в полночь. Если необходимо отобрать заказы с определенной датой, вы можете обойтись без фильтрации по диапазону. Для этой задачи подойдет оператор равенства.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate = '20070212';
```

Преобразовывая строковый литерал в тип `DATETIME`, SQL Server считает, что полночь означает отсутствующее время. Так как все значения в столбце `orderdate` указывают на полночь, запрос вернет все заказы с заданной датой. Чтобы разрешить хранение только тех дат, у которых временная часть равна полночи, воспользуйтесь ограничением `CHECK`.

Если время принимает другое значение, вы можете выполнить фильтрацию по диапазону.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070212'
AND orderdate < '20070213';
```

До выхода SQL Server 2008 для работы только со временем была возможность сохранять все значения с датой 1 января 1900 г. Если строковый литерал содержит лишь значение времени, при преобразовании его в типы DATETIME и SMALLDATETIME эта дата используется как начальная. Например, запустите код, представленный ниже.

```
SELECT CAST('12:30:15.123' AS DATETIME);
```

Вы получите следующий результат.

```
-----  
1900-01-01 12:30:15.123
```

Представьте, что у вас есть таблица со столбцом `tm` типа DATETIME и вы хотите хранить все значения с начальной датой. Как упоминалось выше, этого можно добиться с помощью ограничения CHECK. Чтобы вернуть все записи, для которых время равно 12:30:15.123, используйте фильтр `WHERE tm = '12:30:15.123'`. Преобразовывая строковый литерал в тип DATETIME, SQL Server применит начальную дату, поскольку другого значения вы не указали.

Если вам необходимо разделить дату и время, но входящие значения содержат оба компонента, «обнулите» ненужную часть. Если требуется лишь дата, используйте полночь в качестве времени; если только время, установите дату 1 января 1900 г. В разделе «Функции даты и времени» текущей главы я покажу вам, как это сделать.

## Фильтрация дат по диапазону

Когда необходимо отфильтровать определенный диапазон дат (например, месяц или год), логично использовать функции YEAR и MONTH. Следующий запрос возвращает все заказы, размещенные в 2007 г.

```
SELECT orderid, custid, empid, orderdate  
FROM Sales.Orders  
WHERE YEAR(orderdate) = 2007;
```

Однако вы должны понимать, что чаще всего, работая с фильтруемыми столбцами, SQL Server не обеспечивает эффективного использования индексов. Без общего представления об индексах и производительности (сведения о которых выходят за рамки нашей книги) этот факт не столь очевиден, поэтому пока руководствуйтесь следующим принципом: чтобы получать выгоду от применения индексов, избегайте работы с фильтруемыми столбцами в своих предикатах. Вот как можно переписать предыдущий запрос.

```
SELECT orderid, custid, empid, orderdate  
FROM Sales.Orders  
WHERE orderdate >= '20070101' AND orderdate < '20080101';
```

Аналогичным образом можно отбирать заказы за определенный месяц, не прибегая к функции MONTH. Возьмем для примера следующий запрос.

```
SELECT orderid, custid, empid, orderdate  
FROM Sales.Orders  
WHERE YEAR(orderdate) = 2007 AND MONTH(orderdate) = 2;
```

В нем можно применить фильтрацию по диапазону.

```
SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderdate >= '20070201' AND orderdate < '20070301';
```

## Функции даты и времени

Сейчас я опишу функции, предназначенные для работы с датой и временем: GETDATE, CURRENT\_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME, SYSDATETIMEOFFSET, CAST, CONVERT, SWITCHOFFSET, TODATETIMEOFFSET, DATEADD, DATEDIFF, DATEPART, YEAR, MONTH, DAY, DATENAME, EOMONTH и разные варианты FROMPARTS.

Функции SYSDATETIME, SYSUTCDATETIME, SYSDATETIMEOFFSET, SWITCHOFFSET и TODATETIMEOFFSET появились в SQL Server 2008. В этой версии также доработаны ранее существовавшие функции, которые теперь поддерживают новые типы и элементы. Функция EOMONTH и различные разновидности FROMPARTS были представлены в SQL Server 2012.

## Текущие дата и время

Следующие функции не принимают параметров и возвращают текущие дату и время в системе, в которой работает SQL Server: GETDATE, CURRENT\_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME и SYSDATETIMEOFFSET. Они описаны в таблице 2.3.

**Таблица 2.3.** Функции, которые возвращают текущие дату и время

Функция	Возвращаемый тип	Описание
GETDATE	DATETIME	Текущие дата и время
CURRENT_TIMESTAMP	DATETIME	Аналог GETDATE, совместимый с ANSI SQL
GETUTCDATE	DATETIME	Текущие дата и время в формате UTC
SYSDATETIME	DATETIME2	Текущие дата и время
SYSUTCDATETIME	DATETIME2	Текущие дата и время в формате UTC
SYSDATETIMEOFFSET	DATETIMEOFFSET	Текущие дата и время с учетом часового пояса

Все функции, даже те, которые не принимают аргументов, записываются со скобками (исключение составляет функция CURRENT\_TIMESTAMP, входящая в стандарт ANSI). Функции CURRENT\_TIMESTAMP и GETDATE дают одинаковый результат, при этом первая является стандартной, поэтому желательно использовать именно ее.

Следующий код демонстрирует использование функций, которые возвращают текущие дату и время.

```
SELECT
    GETDATE ()           AS [GETDATE] ,
    CURRENT_TIMESTAMP    AS [CURRENT_TIMESTAMP] ,
    GETUTCDATE ()       AS [GETUTCDATE] ,
    SYSDATETIME ()      AS [SYSDATETIME] ,
    SYSUTCDATETIME ()   AS [SYSUTCDATETIME] ,
    SYSDATETIMEOFFSET () AS [SYSDATETIMEOFFSET];
```

Эти функции выдают комбинированное значение даты и времени. Однако вы легко сможете преобразовать типы `CURRENT_TIMESTAMP` или `SYSDATETIME` в значения `DATE` или `TIME`.

```
SELECT
    CAST(SYSDATETIME() AS DATE) AS [current_date],
    CAST(SYSDATETIME() AS TIME) AS [current_time];
```

## Функции CAST, CONVERT, PARSE и их аналоги с префиксом TRY\_

Функции `CAST`, `CONVERT` и `PARSE` используются, чтобы привести значения к определенным типам. Если преобразование прошло успешно, возвращается результат; в противном случае запрос завершается ошибкой. У каждой из этих функций существуют аналоги с префиксом `TRY_`: `TRY_CAST`, `TRY_CONVERT` и `TRY_PARSE`. Они содержат те же аргументы и выполняют идентичные действия, но в случае неудачного преобразования запрос возвращает значение `NULL`.

Функции `TRY_CAST`, `TRY_CONVERT`, `PARSE` и `TRY_PARSE` появились в SQL Server 2012.

### Синтаксис:

`CAST(значение AS тип)`

`TRY_CAST(значение AS тип)`

`CONVERT (тип, значение [, номер_стиля])`

`TRY_CONVERT (тип, значение [, номер_стиля])`

`PARSE (значение AS тип [USING регион])`

`TRY_PARSE (значение AS тип [USING регион])`

Три основные функции приводят исходное *значение* к заданному *типу*. В некоторых случаях функция `CONVERT` содержит третий аргумент, в котором указывается *стиль* преобразования. Приводя строку к одному из типов даты и времени (или наоборот), вы можете выбрать ее формат. Например, стиль 101 обозначает формат 'MM/DD/YYYY', а номер 103 выглядит как 'DD/MM/YYYY'. Полный перечень стилей и их описание вы найдете в электронном справочнике по SQL Server в разделе «Функции CAST и CONVERT (Transact-SQL)». Таким же образом в функции `PARSE` можно указать региональные настройки, например 'en-US' для американского варианта английского языка или 'ru-RU' для русского.

Как упоминалось ранее, отдельные форматы строковых литералов, которые приводятся к типам даты и времени, зависят от языка. Я рекомендую использовать либо нейтральные форматы, либо функции `CONVERT`/`PARSE` (во втором случае необходимо вручную указать номер стиля или регион). Благодаря этому ваш код всегда будет интерпретирован одинаково, независимо от языка, установленного для текущей сессии.

Заметьте, в отличие от `CONVERT` и `PARSE` функция `CAST` является частью спецификации ANSI SQL, поэтому, если вам не нужно вручную задавать номер стиля или

региональные настройки, лучше использовать именно ее; так код будет максимально приближен к стандарту.

Ниже представлено несколько примеров использования функций `CAST`, `CONVERT` и `PARSE` в сочетании с типами даты и времени. Следующий код превращает строковый литерал `'20090212'` в значение типа `DATE`.

```
SELECT CAST('20090212' AS DATE);
```

Дальше идет запрос, который преобразовывает текущее системное значение даты и времени в тип `DATE`, оставляя только дату.

```
SELECT CAST(SYSDATETIME() AS DATE);
```

Следующий запрос делает то же самое, только оставляет не дату, а время. Итоговое значение будет иметь тип `TIME`.

```
SELECT CAST(SYSDATETIME() AS TIME);
```

Как я уже рекомендовал, если вам необходимо работать отдельно с датой и временем, используя при этом типы `DATETIME` или `SMALLDATETIME` (например, для совместимости с версиями, которые выходили до SQL Server 2008), «обнулите» ненужную часть.

Код, представленный ниже, превращает текущие дату и время в значение типа `CHAR(8)`, используя стиль `112 ('YYYYMMDD')`.

```
SELECT CONVERT(CHAR(8), CURRENT_TIMESTAMP, 112);
```

Если сегодня 12 февраля 2009 г., код вернет `'20090212'`. Этот стиль не зависит от языка, поэтому после обратного преобразования в тип `DATETIME` вы получите ту же дату со временем, установленным в полночь.

```
SELECT CAST(CONVERT(CHAR(8), CURRENT_TIMESTAMP, 112) AS DATETIME);
```

Аналогично, чтобы «обнулить» часть значения, которая касается даты, приведите значение к типу `CHAR(12)`, используя стиль `114 ('hh:mm:ss.nnn')`.

```
SELECT CONVERT(CHAR(12), CURRENT_TIMESTAMP, 114);
```

Выполнив обратное преобразование, вы получите значение типа `DATETIME` с текущим временем и начальной датой.

```
SELECT CAST(CONVERT(CHAR(12), CURRENT_TIMESTAMP, 114) AS DATETIME);
```

Что касается функции `PARSE`, ниже представлено несколько примеров, которые я уже показывал в этой главе.

```
SELECT PARSE('02/12/2007' AS DATETIME USING 'en-US');
```

```
SELECT PARSE('02/12/2007' AS DATETIME USING 'ru-RU');
```

В первом используются американские региональные настройки, во втором — русские.



## Функция SWITCHOFFSET

Корректирует исходное значение типа DATETIMEOFFSET в соответствии с часовым поясом.

### Синтаксис:

```
SWITCHOFFSET(значение_datetimeoffset, часовой_пояс)
```

Приведенный ниже код устанавливает для текущего системного значения `datetimeoffset` часовой пояс `-05:00`.

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '-05:00');
```

Если текущее системное значение `datetimeoffset` равно 12 февраля 2009 г. 10:00:00.0000000 -08:00, код вернет 12 февраля 2009 13:00:00.0000000 -05:00.

Приведенный ниже запрос сдвигает часовой пояс по UTC.

```
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '+00:00');
```

Для вышеупомянутого значения `datetimeoffset` код вернет 12 февраля 2009 г. 18:00:00.0000000 +00:00.

## Функция TODATETIMEOFFSET

Сдвигает часовой пояс к значению даты и времени.

### Синтаксис:

```
TODATETIMEOFFSET(значение_даты_и_времени, часовой_пояс)
```

Эта функция отличается от предыдущей тем, что в качестве ее первого аргумента обычно выступает тип даты и времени, не поддерживающий временной сдвиг. Она просто объединяет исходное значение с указанным часовым поясом, возвращая новый экземпляр `datetimeoffset`.

Обычно эту функцию используют для преобразования данных, которые не поддерживают часовые пояса.

Представьте, что существует таблица под названием `dt`, хранящая локальные значения даты и времени типа `DATETIME`, временной сдвиг в ней представлен отдельным атрибутом `theoffset`. Вам необходимо объединить эти две составляющие в один столбец под названием `dto`. Для этого сначала выполните команду `ALTER TABLE` и добавьте новый атрибут. Затем запишите в нее результаты выражения `TODATETIMEOFFSET(dt, theoffset)` и удалите старые атрибуты `dt` и `theoffset`.

## Функция DATEADD

Добавляет заданный интервал в определенную часть даты и времени.

### Синтаксис:

```
DATEADD(часть, интервал, значение_даты)
```

В качестве первого аргумента поддерживаются значения `year`, `quarter`, `month`, `dayofyear`, `day`, `week`, `weekday`, `hour`, `minute`, `second`, `millisecond`, `microsecond` и `nanosecond`. Можно также использовать сокращенную запись — например, `yy` вместо `year`. Подробности вы найдете в электронном справочнике по SQL Server.

Возвращаемый тип такой же, как у исходного значения даты и времени. Если передать этой функции строковый литерал, она вернет `DATETIME`.

Следующий код добавляет к дате 12 февраля 2009 г. один год.

```
SELECT DATEADD(year, 1, '20090212');
```

В итоге мы получим результат, представленный ниже.

```
-----  
2010-02-12 00:00:00.000
```

## Функция DATEDIFF

Возвращает разницу между двумя значениями даты и времени в определенных единицах (годы, месяцы, дни и т. д.).

### Синтаксис:

```
DATEDIFF(единицы, значение1, значение2)
```

Например, приведенный код возвращает разницу между значениями в днях.

```
SELECT DATEDIFF(day, '20080212', '20090212');
```

Результатом будет число 366.

Задействуем функции `DATEADD` и `DATEDIFF` в более сложном сценарии. Следующий код устанавливает полночь для текущего системного значения даты и времени (работает в версиях, вышедших до SQL Server 2008).

```
SELECT  
    DATEADD(  
        day,  
        DATEDIFF(day, '20010101', CURRENT_TIMESTAMP), '20010101');
```

Сначала функция `DATEDIFF` вычисляет разницу между текущим значением и некоторой фиксированной датой с обнуленным временем (в нашем случае это '20010101'). Мы получим определенное количество дней, которое с помощью функции `DATEADD` добавляется к фиксированной дате. В итоге у нас будет текущая системная дата с обнуленным временем.

Если вместо аргументов `day` использовать `month` и взять фиксированную дату с первым числом месяца (как в примере), получится первый день в текущем месяце.

```
SELECT  
    DATEADD(  
        month,  
        DATEDIFF(month, '20010101', CURRENT_TIMESTAMP), '20010101');
```

Таким же образом можно получить первый день в текущем году, если применить аргументы `year` и соответствующую фиксированную дату.

Этот подход используется и для извлечения последнего дня месяца/года. Например, следующее выражение возвращает последнее число текущего месяца.

```
SELECT
    DATEADD (
        month,
        DATEDIFF(month, '19991231', CURRENT_TIMESTAMP), '19991231');
```

В SQL Server 2012 существует более простой способ вернуть последний день месяца. Для этого предусмотрена функция `EOMONTH`, которую я опишу немного позже.

## Функция DATEPART

Возвращает целое число, представляющее определенную часть значения даты и времени.

### Синтаксис:

```
DATEPART(часть, значение)
```

В качестве первого аргумента применяются значения `year`, `quarter`, `month`, `dayofyear`, `day`, `week`, `weekday`, `hour`, `minute`, `second`, `millisecond`, `microsecond`, `nanosecond`, `TZoffset` и `ISO_WEEK`. Последние четыре доступны в SQL Server 2008 и 2012. Как упоминалось ранее, вы можете использовать сокращенную запись — `yy` вместо `year`, `mm` вместо `month`, `dd` вместо `day` и т. д.

Следующий код возвращает количество месяцев в заданной дате.

```
SELECT DATEPART(month, '20090212');
```

Результат равен числу 2.

## Функции YEAR, MONTH и DAY

`YEAR`, `MONTH` и `DAY` — упрощенные версии функции `DATEPART`. Они возвращают целое число, которое представляет определенную часть (год, месяц или день) значения даты и времени.

### Синтаксис:

```
YEAR(значение)
```

```
MONTH(значение)
```

```
DAY(значение)
```

Например, приведенный код извлекает из исходной даты день, месяц и год.

```
SELECT
    DAY('20090212') AS theday,
    MONTH('20090212') AS themonth,
    YEAR('20090212') AS theyear;
```

Ниже представлен результат.

theday	themonth	theyear
-----	-----	-----
12	2	2009

## Функция DATENAME

Возвращает строку с определенной частью значения даты и времени.

### Синтаксис:

DATENAME(значение, часть)

Здесь в качестве части даты выступают те же параметры, что и в функции DATEPART. Однако значение указанной части возвращается в виде названия, а не числа. Следующий код генерирует название месяца для заданной даты.

```
SELECT DATENAME(month, '20090212');
```

Как вы помните, для этого входного значения функция DATEPART вернула число 2. Функция DATENAME возвращает название месяца, которое зависит от языковых настроек. Если в вашей сессии в качестве языка установлена одна из версий английского (us\_english или British), вы получите строку 'February'. Для русского языка значение выглядит как 'февраль'. Если вы запрашиваете часть даты, у которой существует только числовое представление (например, год), функция DATENAME возвращает число в виде строки. Например, следующий код вернет '2009':

```
SELECT DATENAME(year, '20090212');
```

## Функция ISDATE

Принимает в качестве аргумента строку и, если ее можно привести к типу даты и времени, возвращает 1; в противном случае — 0.

### Синтаксис:

ISDATE(строка)

Результатом выполнения следующего кода будет 1.

```
SELECT ISDATE('20090212');
```

Следующий запрос вернет 0.

```
SELECT ISDATE('20090230');
```

## Функции \*FROMPARTS

Представлены в SQL Server 2012. Они принимают целые числа, представляющие отдельные части даты и времени, и генерируют на их основе значение соответствующего типа.

**Синтаксис:**

DATEFROMPARTS (год, месяц, день)

DATETIME2FROMPARTS (год, месяц, день, час, минута, секунды, доли\_секунды, точность)

DATETIMEFROMPARTS (год, месяц, день, час, минута, секунды, миллисекунды)

DATETIMEOFFSETFROMPARTS (год, месяц, день, час, минута, секунды, доли\_секунды, часовой\_сдвиг, минутный\_сдвиг, точность)

SMALLDATETIMEFROMPARTS (год, месяц, день, час, минута)

TIMEFROMPARTS (час, минута, секунды, доли\_секунды, точность)

Функции упрощают программное создание значений даты и времени на основе отдельных составляющих. Они появились в SQL Server, чтобы улучшить совместимость с другими СУРБД, которые обладают похожими возможностями. Продемонстрируем в следующем запросе, как применять эти функции.

```
SELECT
    DATEFROMPARTS(2012, 02, 12),
    DATETIME2FROMPARTS(2012, 02, 12, 13, 30, 5, 1, 7),
    DATETIMEFROMPARTS(2012, 02, 12, 13, 30, 5, 997),
    DATETIMEOFFSETFROMPARTS(2012, 02, 12, 13, 30, 5, 1, -8, 0, 7),
    SMALLDATETIMEFROMPARTS(2012, 02, 12, 13, 30),
    TIMEFROMPARTS(13, 30, 5, 1, 7);
```

**Функция EOMONTH**

Также появилась в SQL Server 2012. В качестве аргумента она принимает дату и время и возвращает значение типа DATE, представляющее полночь последнего дня соответствующего месяца. Существует и необязательный аргумент, посредством которого можно добавить определенное количество месяцев.

**Синтаксис:**

EOMONTH (дата [, дополнительные\_месяцы])

Следующий код возвращает конец текущего месяца.

```
SELECT EOMONTH(SYSDATETIME());
```

А этот запрос вернет заказы, сделанные в последний день месяца.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate = EOMONTH(orderdate);
```

**Извлечение метаданных**

SQL Server содержит инструменты, которые позволяют получить метаданные таких объектов, как таблицы в БД и столбцы в таблице. Эти инструменты

включают представления каталога и информационной схемы, сохраненные в системе процедуры и функции. Данная тема хорошо освещена в разделе «Запрос к системному каталогу сервера SQL Server» электронного справочника, поэтому здесь мы не будем углубляться в подробности. Я приведу несколько примеров, чтобы в общих чертах рассказать о доступных вам инструментах и их возможностях.

### Представления каталога

С помощью представлений каталога вы получите подробную информацию об объектах, которые хранятся в БД, в том числе сведения, касающиеся SQL Server. Например, чтобы перечислить все таблицы в БД вместе с названиями схемы, сделайте запрос к представлению `sys.tables`.

```
USE TSQL2012;

SELECT SCHEMA_NAME(schema_id) AS table_schema_name, name AS table_name
FROM sys.tables;
```

Функция `SCHEMA_NAME` преобразовывает целочисленный идентификатор схемы в строковое название. Запрос возвращает следующий результат.

table_schema_name	table_name
-----	-----
HR	Employees
Production	Suppliers
Production	Categories
Production	Products
Sales	Customers
Sales	Shippers
Sales	Orders
Sales	OrderDetails
Stats	Tests
Stats	Scores
dbo	Nums

Информация о столбцах таблицы находится в представлении `sys.columns`. С помощью кода, приведенного ниже, вы получите данные о столбцах таблицы `Sales.Orders`, включая название, тип (системный идентификатор типа транслируется в строковое название с помощью функции `TYPE_NAME`), максимальную длину, параметры сравнения и сведения о поддержке отметок типа `NULL`.

Вот какой результат вы получите.

column_name	column_type	max_length	collation_name	is_nullable
-----	-----	-----	-----	-----
orderid	int	4	NULL	0
custid	int	4	NULL	1
empid	int	4	NULL	0
orderdate	datetime	8	NULL	0
requireddate	datetime	8	NULL	0
shippeddate	datetime	8	NULL	1
shipperid	int	4	NULL	0
freight	money	8	NULL	0
shipname	nvarchar	120	Cyrillic_General_CI_AS	0

shipaddress	nvarchar	120	Cyrillic_General_CI_AS	0
shipcity	nvarchar	120	Cyrillic_General_CI_AS	0
shipregion	nvarchar	120	Cyrillic_General_CI_AS	1
shippostalcode	nvarchar	120	Cyrillic_General_CI_AS	1
shipcountry	nvarchar	120	Cyrillic_General_CI_AS	0

## Представления информационной схемы

Информационная схема (INFORMATION\_SCHEMA) состоит из нескольких представлений и позволяет получить метаданные, которые соответствуют стандарту SQL (то есть не касаются фирменных особенностей SQL Server).

Ниже показан запрос к представлению INFORMATION\_SCHEMA.TABLES. Он перечисляет пользовательские таблицы в текущей БД вместе с названиями их схемы.

```
SELECT TABLE_SCHEMA, TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = N'BASE TABLE';
```

Следующий запрос обращается к представлению INFORMATION\_SCHEMA.COLUMNS, возвращая бо́льшую часть информации о столбцах в таблице Sales.Orders.

```
SELECT
    COLUMN_NAME, DATA_TYPE, CHARACTER_MAXIMUM_LENGTH,
    COLLATION_NAME, IS_NULLABLE
FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_SCHEMA = N'Sales'
    AND TABLE_NAME = N'Orders';
```

## Системные хранимые процедуры и функции

Системные хранимые процедуры и функции выполняют внутренние запросы к каталогу, предоставляя систематизированный набор метаданных. Полный список объектов и их описание вы найдете в электронном справочнике по SQL Server, а пока рассмотрим несколько примеров.

Хранимая процедура sp\_tables возвращает список объектов (таблиц и представлений), к которым можно получить доступ в текущей БД.

```
EXEC sys.sp_tables;
```

Процедура sp\_help принимает в качестве аргумента название объекта и возвращает информацию о нем, его столбцах, индексах, ограничениях и т. д. Следующий код вернет несколько результирующих наборов с подробными данными о таблице Orders.

```
EXEC sys.sp_help
    @objname = N'Sales.Orders';
```

Процедура sp\_columns возвращает информацию о столбцах объекта. Получим, к примеру, сведения о столбцах таблицы Orders.

```
EXEC sys.sp_columns
    @table_name = N'Orders',
    @table_owner = N'Sales';
```

Процедура `sp_helpconstraint` возвращает информацию об ограничениях объекта. С помощью следующего кода вы узнаете ограничения таблицы `Orders`.

```
EXEC sys.sp_helpconstraint
    @objname = N'Sales.Orders';
```

В вашем распоряжении находится также набор функций, которые выдают сведения об экземплярах SQL Server, БД, объектах, столбцах и т. д. Функция `SERVERPROPERTY` возвращает заданное свойство текущего экземпляра. Используя приведенный код, можно узнать уровень версии (RTM, SP1, SP2 и т.д.) SQL Server.

```
SELECT
    SERVERPROPERTY('ProductLevel');
```

Функция `DATABASEPROPERTYEX` возвращает одно из свойств заданной БД. Например, чтобы получить параметры сравнения БД `TSQL2012`, запустите следующий код.

```
SELECT
    DATABASEPROPERTYEX(N'TSQL2012', 'Collation');
```

Функция `OBJECTPROPERTY` возвращает свойство указанного объекта. Код, представленный ниже, показывает, есть ли у таблицы `Orders` первичный ключ.

```
SELECT
    OBJECTPROPERTY(OBJECT_ID(N'Sales.Orders'), 'TableHasPrimaryKey');
```

В этом запросе можно наблюдать интересную вложенность. Функция `OBJECTPROPERTY` принимает числовой идентификатор, а не название таблицы. Чтобы его получить, пришлось использовать функцию `OBJECT_ID`.

Функция `COLUMNPROPERTY` возвращает свойство заданного столбца. Например, в результате выполнения следующего кода можно выяснить, поддерживает ли столбец `shipcountry` в таблице `Orders` отметки типа `NULL`.

```
SELECT
    COLUMNPROPERTY(OBJECT_ID(N'Sales.Orders'), N'shipcountry',
    'AllowsNull');
```

## В заключение

В этой главе вы познакомились с инструкцией `SELECT`, логическим порядком обработки и другими аспектами однотабличных запросов. Мы рассмотрели много тем и узнали о новых концепциях. Если вы пока не понимаете некоторые моменты, то всегда сможете вернуться к ним позже.



## Упражнения

Здесь собраны упражнения, которые помогут вам усвоить материал главы. Ответы находятся в следующем разделе.

### 1

Напишите запрос к таблице **Sales.Orders**, который возвращает заказы, размещенные в июне 2007 г.

Используются БД **TSQL2012** и таблица **Sales.Orders**.

Ожидаемый результат (в сокращенном виде):

orderid	orderdate	custid	empid
-----	-----	-----	-----
10555	2007-06-02 00:00:00.000	71	6
10556	2007-06-03 00:00:00.000	73	2
10557	2007-06-03 00:00:00.000	44	9
10558	2007-06-04 00:00:00.000	4	1
10559	2007-06-05 00:00:00.000	7	6
10560	2007-06-06 00:00:00.000	25	8
10561	2007-06-06 00:00:00.000	24	2
10562	2007-06-09 00:00:00.000	66	1
10563	2007-06-10 00:00:00.000	67	2
10564	2007-06-10 00:00:00.000	65	4
...			
(строк обработано: 30)			

### 2

Напишите запрос к таблице **Sales.Orders**, который возвращает заказы, размещенные в последний день месяца.

Используются БД **TSQL2012** и таблица **Sales.Orders**.

Ожидаемый результат (в сокращенном виде):

orderid	orderdate	custid	empid
-----	-----	-----	-----
10269	2006-07-31 00:00:00.000	89	5
10317	2006-09-30 00:00:00.000	48	6
10343	2006-10-31 00:00:00.000	44	4
10399	2006-12-31 00:00:00.000	83	8
10432	2007-01-31 00:00:00.000	75	3
10460	2007-02-28 00:00:00.000	24	8
10461	2007-02-28 00:00:00.000	46	1
10490	2007-03-31 00:00:00.000	35	7
10491	2007-03-31 00:00:00.000	28	8
10522	2007-04-30 00:00:00.000	44	4
...			
(строк обработано: 26)			

**3**

Напишите запрос к таблице **HR.Employees**, который возвращает записи о сотрудниках, чьи фамилии содержат больше одной буквы «о».

Используются БД **TSQL2012** и таблица **HR.Employees**.

Ожидаемый результат:

empid	firstname	lastname
9	Зоя	Долгопятова

(строк обработано: 1)

**4**

Напишите запрос к таблице **Sales.OrderDetails**, который возвращает заказы с общей стоимостью (вычисляется как количество, умноженное на цену отдельного товара) более 10 000. Сортировка выполняется по общей стоимости.

Используются БД **TSQL2012** и таблица **Sales.OrderDetails**.

Ожидаемый результат:

orderid	totalvalue
10865	17250,00
11030	16321,90
10981	15810,00
10372	12281,20
10424	11493,20
10817	11490,70
10889	11380,00
10417	11283,20
10897	10835,24
10353	10741,60
10515	10588,50
10479	10495,60
10540	10191,70
10691	10164,80

(строк обработано: 14)

**5**

Напишите запрос к таблице **Sales.Orders**, чтобы он возвращал три страны, средняя стоимость отправки заказов в которые была самой высокой в 2007 г.

Используются БД **TSQL2012** и таблица **Sales.Orders**.

Ожидаемый результат:

shipcountry	avgfreight
Австрия	178,3642
Швейцария	117,1775
Швеция	105,16

(строк обработано: 3)

## 6

Напишите запрос к таблице `Sales.Orders`, который вычисляет номера строк отдельно для каждого клиента, исходя из даты размещения заказа (примените идентификатор заказа в качестве дополнительного условия).

Используются БД `TSQL2012` и таблица `Sales.Orders`.

Ожидаемый результат (в сокращенном виде):

custid	orderdate	orderid	rownum
1	2007-08-25 00:00:00.000	10643	1
1	2007-10-03 00:00:00.000	10692	2
1	2007-10-13 00:00:00.000	10702	3
1	2008-01-15 00:00:00.000	10835	4
1	2008-03-16 00:00:00.000	10952	5
1	2008-04-09 00:00:00.000	11011	6
2	2006-09-18 00:00:00.000	10308	1
2	2007-08-08 00:00:00.000	10625	2
2	2007-11-28 00:00:00.000	10759	3
2	2008-03-04 00:00:00.000	10926	4
...			

(строк обработано: 830)

## 7

Используя таблицу `HR.Employees`, создайте инструкцию `SELECT`, которая определяет пол каждого сотрудника, исходя из формы обращения к нему. Значения 'мисс' и 'миссис' означают 'женщина'; 'мистер' означает 'мужчина'; во всех остальных случаях (например, 'доктор') должно возвращаться 'неизвестно'.

Используются БД `TSQL2012` и таблица `HR.Employees`.

Ожидаемый результат:

empid	firstname	lastname	titleofcourtesy	gender
1	Сара	Дэвис	мисс	женщина
2	Дон	Функ	доктор	неизвестно
3	Джуди	Лью	мисс	женщина
4	Иаиль	Пелед	миссис	женщина
5	Свен	Бак	мистер	мужчина
6	Пол	Суурс	мистер	мужчина
7	Рассел	Кинг	мистер	мужчина
8	Мария	Камерон	мисс	женщина
9	Зоя	Долгопятова	мисс	женщина

(строк обработано: 9)

## 8

Напишите запрос к таблице `Sales.Customers`, который возвращает для каждого клиента его идентификатор и регион. Сортировка должна выполняться по региону, а отметки `NULL` находиться после «ненулевых» значений. Помните, что по умолчанию язык `T-SQL` размещает отметки `NULL` в начале результирующего набора.

Используются БД **TSQL2012** и таблица **Sales.Customers**.

Ожидаемый результат (в сокращенном виде):

custid	region
55	AK
10	BC
42	BC
45	CA
33	DF
71	ID
78	MT
65	NM
32	OR
...	
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL
11	NULL
...	

(строк обработано: 91)

## Решения

Приведем решения для упражнений из этой главы. При необходимости дадим пояснения.

### 1

Вы можете применить функции **YEAR** и **MONTH** внутри инструкции **SELECT**.

```
USE TSQL2012;

SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE YEAR(orderdate) = 2007 AND MONTH(orderdate) = 6;
```

Это приемлемое решение, и его результат является корректным. Однако в случае манипуляций с фильтруемым столбцом **SQL Server**, скорее всего, не сможет повысить эффективность запроса посредством индексов, поэтому я рекомендую использовать фильтрацию по диапазону.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate >= '20070601'
      AND orderdate < '20070701';
```

## 2

Для решения этой задачи в SQL Server 2012 можно использовать функцию EOMONTH.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate = EOMONTH(orderdate);
```

В предыдущих версиях SQL Server все немного сложнее. Когда мы рассматривали функции даты и времени, я приводил выражение, которое вычисляет последний день месяца в соответствии с указанной датой.

```
DATEADD(month, DATEDIFF(month, '19991231', date_val), '19991231')
```

Здесь вычисляется разница между фиксированной (31 декабря 1999 г.) и заданной датами (в месяцах). Затем это значение добавляется к фиксированной дате и в результате мы получаем последний день месяца для указанной даты. Ниже представлено решение, которое вернет только заказы, размещенные в последний день месяца.

```
SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate = DATEADD(month, DATEDIFF(month, '19991231', orderdate),
'19991231');
```

## 3

В данном упражнении необходимо использовать поиск по шаблону с помощью предиката LIKE. Как вы помните, знак процента (%) представляет строку любой длины (в том числе и нулевой). Таким образом, чтобы выразить вхождение как минимум двух символов «о», укажите шаблон '%о%о%'. Приведем решение.

```
SELECT empid, firstname, lastname
FROM HR.Employees
WHERE lastname LIKE '%о%о%';
```

## 4

Это упражнение с подвохом. Если вы решили его правильно, можете собой гордиться. К запросу выдвигается не совсем очевидное требование, которое можно не заметить или неправильно интерпретировать. В условии сказано: «возвращает заказы, чья общая стоимость превышает 10 000». Обратите внимание на слово «общая». Такое требование касается всей группы заказов, а не отдельных записей. То есть в инструкции WHERE не должен содержаться фильтр наподобие представленного ниже.

```
WHERE quantity * unitprice > 10000
```

Вместо этого запрос должен группировать данные по идентификатору заказа, фильтруя записи внутри инструкции HAVING.

---

```
HAVING SUM(quantity*unitprice) > 10000
```

Вот решение.

```
SELECT orderid, SUM(qty*unitprice) AS totalvalue
FROM Sales.OrderDetails
GROUP BY orderid
HAVING SUM(qty*unitprice) > 10000
ORDER BY totalvalue DESC;
```

## 5

Поскольку нас интересуют заказы за 2007 г., запрос должен содержать инструкцию WHERE с соответствующим фильтром по диапазону дат (orderdate >= '20070101' AND orderdate < '20080101'). Нам требуется средняя стоимость отправки для каждой страны, при этом страны могут быть представлены несколькими записями. Таким образом, перед вычислением средней стоимости отправки необходимо сгруппировать строки по столбцу **shipcountry**. Чтобы получить три страны с самой дорогой доставкой, воспользуемся выражением TOP (3), отсортировав строки по средней стоимости. Ниже показано решение.

```
SELECT TOP (3) shipcountry, AVG(freight) AS avgfreight
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate < '20080101'
GROUP BY shipcountry
ORDER BY avgfreight DESC;
```

В SQL Server 2012 вместо фирменного выражения TOP можно использовать стандартный параметр OFFSET-FETCH. В этом случае решение выглядит следующим образом.

```
SELECT shipcountry, AVG(freight) AS avgfreight
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate < '20080101'
GROUP BY shipcountry
ORDER BY avgfreight DESC
OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY;
```

## 6

Поскольку количество строк необходимо подсчитывать отдельно для каждого клиента, запрос должен содержать инструкцию PARTITION BY custid. Кроме того, требуется, чтобы результат был упорядочен по столбцу **orderdate** с дополнительным условием **orderid**. Следовательно, в инструкции OVER следует указать выражение ORDER BY orderdate, orderid. Приведем решение.

```
SELECT custid, orderdate, orderid,
       ROW_NUMBER() OVER(PARTITION BY custid ORDER BY orderdate, orderid) AS
rownum
FROM Sales.Orders
ORDER BY custid, rownum;
```

## 7

В упражнении условие должно проверяться с помощью простой формы выражения CASE. Для этого после ключевого слова CASE необходимо указать атрибут `titleofcourtesy`. Все формы обращения перечисляются как отдельные инструкции WHEN, а инструкции THEN используются для обозначения пола. Чтобы предусмотреть вариант со значением 'неизвестно', в конце вводится инструкция ELSE.

```
SELECT empid, firstname, lastname, titleofcourtesy,
       CASE titleofcourtesy
         WHEN 'мисс'      THEN 'женщина'
         WHEN 'миссис'   THEN 'женщина'
         WHEN 'мистер'   THEN 'мужчина'
         ELSE              'неизвестно'
       END AS gender
FROM HR.Employees;
```

Вы также можете воспользоваться поисковой формой выражения CASE с двумя предикатами — для женского и мужского пола соответственно. Инструкция ELSE будет отвечать за значения 'неизвестно'.

```
SELECT empid, firstname, lastname, titleofcourtesy,
       CASE
         WHEN titleofcourtesy IN('мисс', 'миссис') THEN 'женщина'
         WHEN titleofcourtesy = 'мистер'          THEN 'мужчина'
         ELSE                                       'неизвестно'
       END AS gender
FROM HR.Employees;
```

## 8

При сортировке SQL Server по умолчанию размещает отметки NULL в начале результирующего набора. Чтобы передвинуть их в конец, примените выражение CASE. Оно вернет 1 для отметок NULL и 0 для остальных значений (что сделает их приоритетными во время сортировки). Результат будет использован как первый сортировочный столбец; вторым выступит столбец **region**. Таким образом, значения, не равные NULL, отсортируются так, как нужно. Далее представлено решение.

```
SELECT custid, region
FROM Sales.Customers
ORDER BY
       CASE WHEN region IS NULL THEN 1 ELSE 0 END, region;
```

## Глава 3

# СОЕДИНЕНИЯ

При выполнении запроса первой обрабатывается инструкция `FROM`, внутри которой с помощью специальных операторов происходит обращение к таблицам. Microsoft SQL Server поддерживает четыре табличных оператора: `JOIN`, `APPLY`, `PIVOT` и `UNPIVOT`. Первый из них является стандартным, тогда как остальные три относятся исключительно к языку T-SQL. Все эти операторы работают с таблицами как с входящими данными, применяя к ним различные этапы логической обработки и возвращая результирующий набор (также в виде таблицы). В этой главе мы сосредоточим свое внимание на табличном операторе `JOIN`. Оператор `APPLY` будет рассмотрен в главе 5, а `PIVOT` и `UNPIVOT` — в главе 7.

Табличный оператор `JOIN` принимает на вход две таблицы. Он может выполнять операции трех различных видов соединения: перекрестного, внутреннего и внешнего. Все они состоят из разных логических этапов обработки. Перекрестное соединение включает только декартово произведение. Внутреннее состоит из декартового произведения и фильтрации. Внешнее соединение, в дополнение к внутреннему, добавляет внешние строки. В этой главе мы подробно рассмотрим каждую из разновидностей соединений и все этапы, через которые они проходят.

Процесс обработки запроса состоит из последовательности логических этапов, которые возвращают корректный результат. На самом деле ядро СУРБД обрабатывает запросы в совсем другом порядке. Некоторые логические этапы могут показаться вам неэффективными, однако на практике они оптимизируются. Нужно сделать акцент на слово «логический», так как операции, которые применяются к исходным таблицам, основываются на реляционной алгебре. Ядро БД не должно строго соблюдать все теоретические постулаты; достаточно, чтобы выдаваемый им результат соответствовал тому, что вытекает из логического процесса обработки. SQL Server имеет много оптимизаций, противоречащих реляционной модели, но это никак не влияет на корректность результата. Несмотря на то что данная книга в основном посвящена логическим аспектам выполнения запросов, я хочу акцентировать ваше внимание на этих обстоятельствах, чтобы избежать недопонимания в дальнейшем.

## Перекрестные соединения

С точки зрения логики перекрестное соединение является простейшим видом оператора `JOIN`. Оно реализует всего один этап обработки — декартово произведение. В нем участвуют две таблицы, которые в результате сопоставления строк превращаются в одну. Таким образом, если в первой таблице  $m$  строк, а во второй  $n$ , то результат будет содержать  $m \times n$  строк.



SQL Server поддерживает две стандартные синтаксические формы перекрестного соединения: ANSI SQL-92 и ANSI SQL-89. Я советую использовать синтаксис ANSI-SQL 92 — именно он применяется во всех примерах данной книги. Чуть позже объясню причину своего выбора, но для полноты изложения в этом разделе будут описаны обе формы.

## Синтаксис ANSI SQL-92

Ниже представлен запрос, который выполняет перекрестное соединение таблиц **Customers** и **Employees** (с помощью синтаксиса ANSI SQL-92) в БД **TSQL2012**, возвращая в результирующем наборе атрибуты **custid** и **empid**.

```
USE TSQL2012;

SELECT C.custid, E.empid
FROM Sales.Customers AS C
      CROSS JOIN HR.Employees AS E;
```

В таблице **Customers** хранится 91 строка, а в таблице **Employees** — 9, поэтому результат запроса будет состоять из 819 записей (см. ниже).

custid	empid
1	2
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	2
11	2
12	2
13	2
14	2
...	

(строк обработано: 819)

Синтаксис ANSI SQL-92 предусматривает наличие ключевых слов **CROSS JOIN** между двумя исходными таблицами.

Обратите внимание, что в приведенном запросе в инструкции **FROM** для таблиц **Customers** и **Employees** были назначены псевдонимы **C** и **E** соответственно. В качестве результирующего набора выступает виртуальная таблица, состоящая из обеих сторон перекрестного соединения. В качестве префиксов для названий столбцов используются назначенные мною псевдонимы (например, **C.custid** и **E.empid**). Если не использовать псевдонимы, в результате будут значиться полные имена исходных таблиц (например, **Customers.custid** и **Employees.empid**). Префиксы нужны для однозначной идентификации столбцов, имена которых могут совпадать. Псевдонимы присваиваются для того, чтобы сделать запрос более лаконичным. Стоит заметить, что использование префиксов является обязательным только в том случае, если в нескольких таблицах присутствуют столбцы с одним и тем же именем. Впрочем, их можно указывать в любом случае — существует мнение, что это помогает

сделать код более понятным. Также нужно понимать, что если у таблиц уже есть псевдонимы, их полные имена нельзя использовать в качестве префиксов; в случае неоднозначности следует указывать именно псевдоним.

## Синтаксис ANSI SQL-89

SQL Server поддерживает более старый вариант синтаксиса для перекрестных соединений, который был представлен в стандарте ANSI SQL-89. Подразумевается, что имена таблиц просто нужно разделить запятыми, как показано ниже.

```
SELECT C.custid, E.empid
FROM Sales.Customers AS C, HR.Employees AS E;
```

Эта версия не отличается от предыдущей ни логикой выполнения, ни производительностью. Оба варианта синтаксиса являются неотъемлемой частью текущего стандарта SQL (на момент написания данной книги это ANSI SQL:2011) и полностью поддерживаются в последней версии SQL Server (Microsoft SQL Server 2012). Нет информации, что синтаксис ANSI SQL-89 планируют объявить устаревшим, и я не вижу для этого ни одной причины. Тем не менее рекомендую использовать синтаксис ANSI SQL-92; вы поймете, почему я сделал такой выбор, когда мы рассмотрим внутренние соединения.

## Перекрестные самосоединения

Вы можете соединить несколько экземпляров одной и той же таблицы. Этот процесс называется **самосоединением** и поддерживается во всех видах оператора JOIN (перекрестном, внутреннем и внешнем). Например, следующий запрос выполняет самосоединение двух экземпляров таблицы **Employees**.

```
SELECT
    E1.empid, E1.firstname, E1.lastname,
    E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
    CROSS JOIN HR.Employees AS E2;
```

В качестве результата мы получим все возможные комбинации из двух сотрудников. Поскольку таблица **Employees** содержит 9 строк, запрос вернет 81 запись (см. ниже).

empid	firstname	lastname	empid	firstname	lastname
1	Сара	Дэвис	1	Сара	Дэвис
2	Дон	Функ	1	Сара	Дэвис
3	Джуди	Лью	1	Сара	Дэвис
4	Иаиль	Пелед	1	Сара	Дэвис
5	Свен	Бак	1	Сара	Дэвис
6	Пол	Суурс	1	Сара	Дэвис
7	Рассел	Кинг	1	Сара	Дэвис
8	Мария	Камерон	1	Сара	Дэвис
9	Зоя	Долгопятова	1	Сара	Дэвис
1	Сара	Дэвис	2	Дон	Функ
2	Дон	Функ	2	Дон	Функ
3	Джуди	Лью	2	Дон	Функ

4	Иаиль	Пелед	2	Дон	Функ
5	Свен	Бак	2	Дон	Функ
6	Пол	Суурс	2	Дон	Функ
7	Рассел	Кинг	2	Дон	Функ
...					

(строка обработано: 81)

Самосоединение требует обязательного наличия псевдонимов для таблиц. Без них нельзя однозначно идентифицировать имена столбцов в результирующем наборе.

## Создание числовых таблиц

Перекрестные соединения могут быть очень полезными при генерировании результирующего набора, который состоит из последовательности целых чисел (1, 2, 3 и т. д.). Это крайне мощный инструмент, который я использую для решения разных задач. Создание таких последовательностей с помощью перекрестных соединений является чрезвычайно эффективным.

Для начала можно создать таблицу **Digits** со столбцом **digit** и заполнить ее десятью строками с числами от 0 до 9. Для этого выполните в контексте БД **TSQL2012** следующий код.

```
USE TSQL2012;
IF OBJECT_ID('dbo.Digits', 'U') IS NOT NULL DROP TABLE dbo.Digits;
CREATE TABLE dbo.Digits(digit INT NOT NULL PRIMARY KEY);

INSERT INTO dbo.Digits(digit)
VALUES (0), (1), (2), (3), (4), (5), (6), (7), (8), (9);

SELECT digit FROM dbo.Digits;
```

Для заполнения таблицы **Digits** используется команда **INSERT**. Все подробности о ее синтаксисе можно найти в главе 8. Содержимое таблицы **Digits** показано ниже.

```
digit
-----
0
1
2
3
4
5
6
7
8
9
```

Представьте, что вам нужно написать запрос, который генерирует последовательность целых чисел от 1 до 1000. Вы можете выполнить перекрестное соединение трех экземпляров таблицы **Digits**, каждый из которых содержит 10 строк и представляет свою степень десяти (1, 10 и 100). В итоге получится 1000 строк. Чтобы получить числовое значение, необходимо перемножить содержимое таблиц на степень десяти, которую представляет каждый экземпляр, затем сложить результаты и добавить 1. Ниже представлена полная версия этого запроса.

```

SELECT D3.digit * 100 + D2.digit * 10 + D1.digit + 1 AS n
FROM      dbo.Digits AS D1
      CROSS JOIN dbo.Digits AS D2
      CROSS JOIN dbo.Digits AS D3
ORDER BY n;

```

Вот что он вернет (результат представлен в сокращенном виде).

```

n
-----
1
2
3
4
5
6
7
8
9
10
...
998
999
1000
(строк обработано: 1000)

```

Это был всего лишь пример того, как сгенерировать последовательность из 1000 чисел. Если вам нужен более объемный результат, можете указать дополнительные экземпляры таблицы `Digits`. Например, для получения 1 000 000 строк вам понадобится шесть экземпляров.

## Внутренние соединения

Внутреннее соединение состоит из двух логических этапов обработки: сначала оно выполняет декартово произведение двух таблиц (как в перекрестном соединении), а затем фильтрует результат с помощью заданного предиката. Этот вид соединений также имеет две версии синтаксиса: ANSI SQL-92 и ANSI SQL-89.

### Синтаксис ANSI SQL-92

Использование синтаксиса ANSI SQL-92 подразумевает наличие между именами таблиц ключевого слова `INNER JOIN`. На самом деле слово `INNER` можно опустить, поскольку любое соединение по умолчанию является внутренним. Чтобы указать предикат, который будет фильтровать строки (условие соединения), используйте инструкцию `ON`.

Следующий запрос выполняет внутреннее соединение таблиц `Employees` и `Orders` из БД `TSQL2012`; сопоставление сотрудников и заказов происходит на основе предиката `E.empid = O.empid`.

```

USE TSQL2012;

SELECT E.empid, E.firstname, E.lastname, O.orderid

```

```
FROM HR.Employees AS E
      JOIN Sales.Orders AS O
      ON E.empid = O.empid;
```

Ниже представлен результат (в сокращенном виде).

empid	firstname	lastname	orderid
1	Сара	Дэвис	10258
1	Сара	Дэвис	10270
1	Сара	Дэвис	10275
1	Сара	Дэвис	10285
1	Сара	Дэвис	10292
...			
2	Дон	Функ	10265
2	Дон	Функ	10277
2	Дон	Функ	10280
2	Дон	Функ	10295
2	Дон	Функ	10300
...			

(строк обработано: 830)

Суть данного запроса можно переформулировать так: мы сопоставляем все записи о сотрудниках и заказах, у которых совпадает идентификатор сотрудника. Это помогает понять принцип работы соединения. Но к подобному вопросу можно подойти более формально, используя концепции реляционной алгебры: сначала выполняется декартово произведение двух таблиц (9 записей о сотрудниках  $\times$   $\times$  830 записей о заказах = 7470 записей), затем строки фильтруются по предикату `E.empid = O.empid`, в результате чего возвращается 830 записей. Как уже упоминалось ранее, это логический порядок обработки соединения; на практике ядро БД может действовать иначе.

Вспомните предыдущую главу, в которой рассматривалось исчисление предиката в троичной системе. Инструкция `ON` по аналогии с `WHERE` и `HAVING` возвращает только те строки, для которых предикат является истинным (`TRUE`); при любом другом результате выполнения предиката (`FALSE` или `UNKNOWN`) строки отбрасываются.

В БД `TSQL2012` у всех сотрудников есть соответствующие заказы, поэтому в результате попадают все записи из таблицы `Employees`. Если бы у какого-то сотрудника не было заказов, запись о нем была бы отфильтрована.

## Синтаксис ANSI SQL-89

Внутренние соединения, как и перекрестные, могут быть записаны с помощью синтаксиса ANSI SQL-89. Для этого между таблицами нужно поставить запятую и определить условие в инструкции `WHERE` (см. ниже).

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E, Sales.Orders AS O
WHERE E.empid = O.empid;
```

Как видите, эта форма записи не предусматривает наличие инструкции `ON`.

Обе версии синтаксиса являются стандартными, полностью поддерживаются в SQL Server и одинаково интерпретируются ядром СУРБД, поэтому никакой разницы в производительности между ними быть не должно. Но, как вы увидите далее, одна из версий является более безопасной.

## Безопасность внутренних соединений

При использовании соединений я настоятельно рекомендую следовать стандарту ANSI SQL-92, так как в некоторых аспектах он является более безопасным. Допустим, вы написали запрос с внутренним соединением, но по ошибке забыли указать условие. В случае использования синтаксиса ANSI SQL-92 запрос становится некорректным и интерпретатор генерирует ошибку. Попробуйте, например, запустить код, представленный ниже.

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E
JOIN Sales.Orders AS O;
```

Вы получите следующее сообщение.

```
Сообщение 102, уровень 15, состояние 1, строка 3
Неправильный синтаксис около конструкции ";".
```

И хотя из данного текста не сразу понятно, что к ошибке привело отсутствие условия соединения, вы сможете это выяснить и устранить проблему. Если же пропустить условие при использовании синтаксиса ANSI SQL-89, получится корректный запрос, который выполняет перекрестное соединение.

```
SELECT E.empid, E.firstname, E.lastname, O.orderid
FROM HR.Employees AS E, Sales.Orders AS O;
```

Поскольку запрос завершается удачно, логическая ошибка может остаться незамеченной и пользователи вашего приложения получат неправильные результаты. Конечно, вряд ли программист забудет указать условие в таком коротком и простом запросе, но настоящий промышленный код является куда более сложным — он может обращаться ко многим таблицам, содержать различные фильтры и другие элементы. В подобных случаях вероятность того, что вы пропустите условие соединения, возрастает.

Надеюсь, я вас убедил в том, что нужно использовать синтаксис ANSI SQL-92. У вас может возникнуть вопрос: а относится ли то же самое к перекрестным соединениям? Ведь у них не бывает условий, поэтому оба варианта должны быть одинаково хорошими. Я рекомендую везде следовать стандарту ANSI SQL-92, так как это сделает ваш код более последовательным.

К тому же синтаксис ANSI SQL-89 не дает четкого представления о том, с каким типом соединения мы имеем дело — перекрестным или внутренним (во втором случае программист мог случайно пропустить условие).

## Другие разновидности соединений

В этом разделе мы рассмотрим другие виды операций на основе оператора JOIN: множественные, составные и не-эквисоединения.

### Составные соединения

**Составным** называют обычное соединение, предикат которого содержит больше одного атрибута с каждой стороны. Этот вид соединений обычно использует составное отношение первичного и внешнего ключей (отношение, основанное сразу на нескольких атрибутах). Представьте, что внешний ключ таблицы `dbo.Table2` состоит из столбцов `col1` и `col2`, которые ссылаются на одноименные столбцы из таблицы `dbo.Table1`, и теперь вам необходимо написать запрос, соединяющий эти две таблицы на основе отношения первичного и внешнего ключей. Инструкция FROM будет выглядеть следующим образом.

```
FROM dbo.Table1 AS T1
JOIN dbo.Table2 AS T2
ON T1.col1 = T2.col1
AND T1.col2 = T2.col2
```

Рассмотрим более реалистичный пример. Допустим, вы хотите отслеживать изменения, которые вносятся в столбцы таблицы `OrderDetails`. Для этого можно создать дополнительную таблицу под названием `OrderDetailsAudit`.

```
USE TSQL2012;
IF OBJECT_ID('Sales.OrderDetailsAudit', 'U') IS NOT NULL
    DROP TABLE Sales.OrderDetailsAudit;
CREATE TABLE Sales.OrderDetailsAudit
(
    lsn          INT          NOT NULL IDENTITY,
    orderid     INT          NOT NULL,
    productid   INT          NOT NULL,
    dt          DATETIME     NOT NULL,
    loginname   sysname      NOT NULL,
    columnname  sysname      NOT NULL,
    oldval      SQL_VARIANT,
    newval      SQL_VARIANT,
    CONSTRAINT PK_OrderDetailsAudit PRIMARY KEY(lsn),
    CONSTRAINT FK_OrderDetailsAudit_OrderDetails
        FOREIGN KEY(orderid, productid)
        REFERENCES Sales.OrderDetails(orderid, productid)
);
```

Каждая строка в таблице `OrderDetailsAudit` хранит серийный номер (`lsn`), ключ отредактированной записи (`orderid`, `productid`), название отредактированного столбца (`columnname`), старое значение (`oldval`), новое значение (`newval`), время внесения изменений (`dt`) и имя пользователя, который произвел запись (`loginname`). Также имеется внешний ключ (столбцы `orderid` и `productid`), который ссылается на первичный ключ таблицы `OrderDetails` (определен для столбцов `orderid` и `productid`). Допустим, вы уже каким-то образом записываете все изменения, которые вносятся в столбцы таблицы `OrderDetails`.

Вам нужно написать запрос к таблицам `OrderDetails` и `OrderDetailsAudit`, который возвращает информацию обо всех изменениях значений, которые имели место в столбце `qty`. Каждая итоговая строка должна содержать текущее содержимое (берется из таблицы `OrderDetails`) и все остальные значения, которые были записаны до и после внесения изменений (необходимые данные можно найти в таблице `OrderDetailsAudit`). Для этого нужно выполнить соединение на основе отношения первичного и внешнего ключей, как показано ниже.

```
SELECT OD.orderid, OD.productid, OD.qty,
       ODA.dt, ODA.loginname, ODA.oldval, ODA.newval
FROM Sales.OrderDetails AS OD
      JOIN Sales.OrderDetailsAudit AS ODA
        ON OD.orderid = ODA.orderid
        AND OD.productid = ODA.productid
WHERE ODA.columnname = N'qty';
```

Поскольку отношение основано на нескольких атрибутах, условие соединения является составным.

## Не-эквисоединения

Соединение, в условии которого участвует только оператор равенства, называют **эквисоединением**. Условие, которое записывается с помощью любых других операторов, называется **не-эквисоединением**.



### ПРИМЕЧАНИЕ

Стандартная версия языка SQL поддерживает так называемые естественные соединения, которые, в сущности, являются внутренними и основываются на сопоставлении разных столбцов с одним и тем же именем. Например, выражение `T1 NATURAL JOIN T2` соединяет строки между `T1` и `T2`, выполняя сопоставление одноименных столбцов с обеих сторон. В языке T-SQL нет реализации естественных соединений (по крайней мере, в версии SQL Server 2012). Если условие основывается на двоичном операторе (равенства или неравенства), операция называется **тета-соединением**. То есть к этой категории можно отнести как экви-, так и не-эквисоединения.

В качестве примера не-эквисоединения возьмем следующий запрос, который генерирует уникальные пары сотрудников, объединяя два экземпляра таблицы `Employees`.

```
SELECT
    E1.empid, E1.firstname, E1.lastname,
    E2.empid, E2.firstname, E2.lastname
FROM HR.Employees AS E1
      JOIN HR.Employees AS E2
        ON E1.empid < E2.empid;
```

Обратите внимание на предикат, указанный в инструкции `ON`. Цель запроса — сгенерировать уникальные пары сотрудников. При использовании перекрестного соединения результат содержал бы пары, состоящие из одинаковых элементов (например, 1 и 1), а также зеркальные сочетания (такие как 1 вместе с 2 и 2 вместе с 1). У внутреннего соединения есть условие, с помощью которого можно сделать



так, чтобы ключ с левой стороны был меньше ключа, который находится справа, что исключит два вышеописанных случая. Пары с одинаковыми элементами отсеются из-за равенства сторон. Зеркальные сочетания тоже не пройдут проверку: один из вариантов будет отбрасываться, потому что левый ключ должен быть меньше правого. Перекрестное соединение генерирует 81 возможный вариант, 36 из которых являются уникальными и возвращаются нашим запросом.

empid	firstname	lastname	empid	firstname	lastname
1	Сара	Дэвис	2	Дон	Функ
1	Сара	Дэвис	3	Джуди	Лью
2	Дон	Функ	3	Джуди	Лью
1	Сара	Дэвис	4	Иаиль	Пелед
2	Дон	Функ	4	Иаиль	Пелед
3	Джуди	Лью	4	Иаиль	Пелед
1	Сара	Дэвис	5	Свен	Бак
2	Дон	Функ	5	Свен	Бак
3	Джуди	Лью	5	Свен	Бак
4	Иаиль	Пелед	5	Свен	Бак
1	Сара	Дэвис	6	Пол	Суурс
2	Дон	Функ	6	Пол	Суурс
3	Джуди	Лью	6	Пол	Суурс
4	Иаиль	Пелед	6	Пол	Суурс
5	Свен	Бак	6	Пол	Суурс
1	Сара	Дэвис	7	Рассел	Кинг
2	Дон	Функ	7	Рассел	Кинг
3	Джуди	Лью	7	Рассел	Кинг
4	Иаиль	Пелед	7	Рассел	Кинг
5	Свен	Бак	7	Рассел	Кинг
6	Пол	Суурс	7	Рассел	Кинг
1	Сара	Дэвис	8	Мария	Камерон
2	Дон	Функ	8	Мария	Камерон
3	Джуди	Лью	8	Мария	Камерон
4	Иаиль	Пелед	8	Мария	Камерон
5	Свен	Бак	8	Мария	Камерон
6	Пол	Суурс	8	Мария	Камерон
7	Рассел	Кинг	8	Мария	Камерон
1	Сара	Дэвис	9	Зоя	Долгопятова
2	Дон	Функ	9	Зоя	Долгопятова
3	Джуди	Лью	9	Зоя	Долгопятова
4	Иаиль	Пелед	9	Зоя	Долгопятова
5	Свен	Бак	9	Зоя	Долгопятова
6	Пол	Суурс	9	Зоя	Долгопятова
7	Рассел	Кинг	9	Зоя	Долгопятова
8	Мария	Камерон	9	Зоя	Долгопятова

(строк обработано: 36)

Если вы все еще не поняли, как работает этот запрос, попробуйте разобрать его пошагово, взяв меньшее количество сотрудников. Представьте, что таблица **Employees** содержит всего три записи: 1, 2 и 3. Сначала выполним декартово произведение двух экземпляров таблицы.

E1.empid	E2.empid
-----	-----
1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3

После отфильтрации строк на основе предиката `E1.empid < E2.empid` останется всего три пары.

E1.empid	E2.empid
-----	-----
1	2
1	3
2	3

## Множественные соединения

Оператор соединения работает только с двумя таблицами, но один запрос может содержать несколько таких операций. В целом, если в инструкции `FROM` появляется несколько табличных операторов, все они выполняются слева направо. Таким образом, результат выполнения первого оператора является левым операндом для второго, и так дальше по цепочке. Если в инструкции `FOR` находится несколько соединений, две таблицы участвуют только в первом случае, а во всех остальных берется результат, полученный на предыдущем этапе. В целях повышения производительности ядро БД может изменять порядок следования операций перекрестного и внутреннего соединений (чаще всего именно так и происходит); однако на корректность итогового результата это никак не влияет.

Следующий запрос соединяет таблицы `Customers` и `Orders`, сопоставляя клиентов с их заказами, и затем добавляет полученный набор к таблице `OrderDetails`, чтобы получить заказы и их составляющие.

```
SELECT
  C.custid, C.companyname, O.orderid,
  OD.productid, OD.qty
FROM Sales.Customers AS C
  JOIN Sales.Orders AS O
    ON C.custid = O.custid
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

Сокращенная версия результата, генерируемого этим запросом, представлена ниже.

custid	companyname	orderid	productid	qty
-----	-----	-----	-----	-----
85	Клиент ENQZT	10248	11	12
85	Клиент ENQZT	10248	42	10

85	Клиент	ENQZT	10248	72	5
79	Клиент	FAPSM	10249	14	9
79	Клиент	FAPSM	10249	51	40
34	Клиент	IBVRG	10250	41	10
34	Клиент	IBVRG	10250	51	35
34	Клиент	IBVRG	10250	65	15
84	Клиент	NRCSK	10251	22	6
84	Клиент	NRCSK	10251	57	15
...					
(строк обработано: 2155)					

## Внешние соединения

По сравнению с другими разновидностями оператора JOIN внешнее соединение обычно оказывается более сложным для восприятия. Начнем с основ. Если вы хорошо усвоите содержимое следующего подраздела и захотите изучить более сложный материал, то в конце данной главы можно будет ознакомиться с нетривиальными аспектами внешний соединений. Однако эти темы не являются обязательными, и вы всегда сможете вернуться к ним позже, когда будете готовы.

### Общие сведения

Внешние соединения появились в стандарте ANSI SQL-92. В отличие от внутренних и перекрестных у них есть только одна версия синтаксиса: ключевое слово JOIN находится между именами таблиц, а условие обозначается инструкцией ON. Внешнее соединение состоит из тех же двух логических этапов, что и внутреннее (декартово произведение и фильтрация), плюс третий уникальный этап, на котором в результирующий набор добавляются внешние строки.

Чтобы оставить таблицы в результирующем наборе, между их именами указываются ключевые слова LEFT OUTER JOIN, RIGHT OUTER JOIN или FULL OUTER JOIN. Слово OUTER является необязательным. Слова LEFT и RIGHT означают, что в результат попадают строки из соответственно левой или правой таблиц; чтобы сохранить строки из обеих сторон, используется слово FULL. На третьем этапе логической обработки внешнее соединение использует предикат ON, чтобы определить записи, для которых не нашлось соответствия в другой таблице (это и есть внешние строки). Позже данные записи добавляются к тем, что уже были отобраны на двух предыдущих этапах; всем атрибутам с противоположной стороны, для которых не нашлось соответствия, устанавливается значение NULL.

Чтобы лучше понять принцип работы внешнего соединения, рассмотрим пример. Следующий запрос соединяет таблицы Customers и Orders, основываясь на соответствии между идентификаторами клиентов, которые в них хранятся; в качестве результата возвращаются записи о клиентах и сделанных ими заказах. Это левое внешнее соединение, поэтому в результирующий набор попадут клиенты, у которых нет ни одного заказа.

```
SELECT C.custid, C.companyname, O.orderid
FROM Sales.Customers AS C
     LEFT OUTER JOIN Sales.Orders AS O
       ON C.custid = O.custid;
```

Ниже представлены данные, сгенерированные этим запросом (в сокращенном виде).

custid	companyname	orderid
-----	-----	-----
1	Клиент NRZBB	10643
1	Клиент NRZBB	10692
1	Клиент NRZBB	10702
1	Клиент NRZBB	10835
1	Клиент NRZBB	10952
1	Клиент NRZBB	11011
...		
21	Клиент KIDPX	10347
21	Клиент KIDPX	10386
21	Клиент KIDPX	10414
21	Клиент KIDPX	10512
21	Клиент KIDPX	10581
21	Клиент KIDPX	10650
21	Клиент KIDPX	10725
22	Клиент DTDMN	NULL
23	Клиент WVFAF	10408
23	Клиент WVFAF	10480
23	Клиент WVFAF	10634
23	Клиент WVFAF	10763
23	Клиент WVFAF	10789
...		
56	Клиент QNIVZ	10999
56	Клиент QNIVZ	11020
57	Клиент WVAXS	NULL
58	Клиент ANXHT	10322
58	Клиент ANXHT	10354
58	Клиент ANXHT	10474
58	Клиент ANXHT	10502
...		
91	Клиент CCFIZ	10374
91	Клиент CCFIZ	10611
91	Клиент CCFIZ	10792
91	Клиент CCFIZ	10870
91	Клиент CCFIZ	10906
91	Клиент CCFIZ	10998
91	Клиент CCFIZ	11044

(строк обработано: 832)

Два клиента в таблице **Customers** не размещали заказов. Их идентификаторы 22 и 57. Заметьте, что в результирующем наборе атрибуты таблицы **Orders** для данных клиентов равны NULL. Очевидно, что эти две записи были отфильтрованы на втором этапе соединения (основанном на предикате ON), но добавлены на третьем, поскольку все записи из левой таблицы должны оставаться на месте. При внутреннем соединении они бы не попали в конечный результат.

Конечный результат внешнего соединения можно воспринимать как совокупность двух типов строк, которые берутся относительно «сохраняемой» таблицы — внутренних и внешних. Внутренние строки имеют соответствия с другой стороны (с учетом предиката ON), а внешние — нет. Если внешнее соединение возвращает оба вида строк, то внутреннее только один (внутренний, как несложно догадаться).

С внешним соединением связан один момент, который часто является источником заблуждений: как вы думаете, в какой инструкции должно размещаться условие — ON или WHERE? Как видно по строкам сохраняемой таблицы, окончательная фильтрация происходит не в инструкции ON; другими словами, предикат ON отвечает только за сопоставление строк с обеих сторон, не определяя, попадут ли они в результирующий набор. Таким образом, эта инструкция годится для задания промежуточных предикатов. Если вы хотите применить фильтр к строкам, которые уже прошли внешнее соединение, и вам нужно, чтобы данная операция выполнялась в самом конце, укажите свой предикат в инструкции WHERE, которая обрабатывается после этапа FROM, в частности, после обработки всех табличных операторов (к которым относится и оператор JOIN) и добавления внешних строк. К тому же, инструкция WHERE, в отличие от ON, окончательно отбрасывает все лишние записи.

Представьте, что вам нужно вернуть записи только о тех клиентах, которые не разместили ни одного заказа (или, говоря более формально, вам нужно вернуть лишь внешние строки). За основу можно взять предыдущий запрос, добавив в него инструкцию WHERE, которая отбирает исключительно внешние строки. Как вы помните, такие строки можно определить по значениям NULL в атрибутах, которые находятся с противоположной стороны относительно сохраняемой таблицы, — именно на них должен ориентироваться наш фильтр.

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
     LEFT OUTER JOIN Sales.Orders AS O
       ON C.custid = O.custid
WHERE O.orderid IS NULL;
```

Этот запрос вернет только две записи о клиентах. Их идентификаторы 22 и 57.

custid	companyname
22	Клиент DTDMM
57	Клиент WVAXS

(строк обработано: 2)

В этом запросе есть несколько моментов, на которые стоит обратить внимание. Уже упоминалось, что для поиска отметок NULL вместо знака равенства необходимо использовать оператор IS NULL. Это вызвано тем, что при сопоставлении отметки NULL с любым другим значением (включая NULL) знак равенства всегда возвращает UNKNOWN. Кроме того, важен атрибут, используемый для фильтрации, который берется из той таблицы, что не полностью попадает в результирующий набор. Нужно отдать предпочтение атрибуту, который во внешней строке может содержать исключительно отметку NULL (другие значения, берущиеся из основной таблицы, не допускаются). Таким образом, у нас есть три варианта: столбец первичного ключа, столбец, по которому выполняется соединение, и столбец, определенный с помощью ограничения NOT NULL. Первичный ключ не может равняться NULL; если его атрибут содержит такое значение, это свидетельствует только об одном — строка является внешней. Если атрибут, по которому выполняется соединение, содержит NULL, строка отбрасывается на втором этапе, когда применяется предикат; иными словами, это также будет признаком внешней строки. То же самое можно сказать о столбце с ограничением NOT NULL, которому присваивается NULL.

Чтобы применить на практике полученные знания и закрепить пройденный материал, не забудьте выполнить упражнения, приведенные в конце главы.

## Дополнительные сведения

Этот раздел посвящен более сложным аспектам внешних соединений. Он не является обязательным к прочтению, и вы можете вернуться к нему позже, когда хорошо усвоите основы.

### Добавление отсутствующих значений

Используя внешние соединения, можно идентифицировать и добавлять отсутствующие значения. Представьте, что вам необходимо получить все заказы из таблицы **Orders** в БД **TSQL2012**. Каждая дата с 1 января 2006 г. по 31 декабря 2008 г. должна иметь как минимум одну запись. Вам не нужно делать ничего особенного с этим диапазоном дат, но результат должен содержать даже те дни, в которые не было сделано ни одного заказа (в данном случае в качестве заглушки вставляется значение **NULL**).

Чтобы решить эту задачу, следует для начала написать запрос, который возвращает последовательность всех дат в нужном диапазоне. Затем вы сможете выполнить внешнее соединение этого набора с таблицей **Orders**. Результат, полученный таким образом, будет включать даты с отсутствующими заказами.

Для получения набора дат в заданном диапазоне я обычно использую вспомогательную таблицу с числами. Эта таблица (назовем ее **dbo.Nums**) имеет всего один столбец **n**, заполненный последовательностью целочисленных значений (1, 2, 3 и т. д.). Я нахожу такой подход чрезвычайно удобным и использую его для решения многим проблем. Достаточно один раз создать подобную таблицу и наполнить ее последовательностью необходимой длины (в БД **TSQL2012** она уже есть).

Первым делом вам необходимо сгенерировать последовательность всех дат в заданном диапазоне. Для этого можно выполнить запрос к таблице **Nums** и отобрать нужное количество записей (по одной на каждый день). Количество дней легко определить с помощью функции **DATEDIFF**. Добавляя  $n - 1$  дней к начальному значению (1 января 2006 г.), вы можете извлечь из последовательности необходимую дату. Вот как выглядит итоговый запрос.

```
SELECT DATEADD(day, n-1, '20060101') AS orderdate
FROM dbo.Nums
WHERE n <= DATEDIFF(day, '20060101', '20081231') + 1
ORDER BY orderdate;
```

В результате возвращается последовательность всех дат с 1 января 2006 г. по 31 декабря 2008 г. (см. ниже).

```
orderdate
-----
2006-01-01 00:00:00.000
2006-01-02 00:00:00.000
2006-01-03 00:00:00.000
2006-01-04 00:00:00.000
2006-01-05 00:00:00.000
...
```

```

2008-12-27 00:00:00.000
2008-12-28 00:00:00.000
2008-12-29 00:00:00.000
2008-12-30 00:00:00.000
2008-12-31 00:00:00.000

```

(строк обработано: 1096)

Теперь добавим в запрос левое внешнее соединение таблиц **Nums** и **Orders**. Условие соединения сопоставляет дату из левой стороны (**Nums**) и значение столбца **orderdate** справа (**Orders**). Для этого используется выражение **DATEADD(day, Nums.n - 1, '20060101')**.

```

SELECT DATEADD(day, Nums.n - 1, '20060101') AS orderdate,
       O.orderid, O.custid, O.empid
FROM   dbo.Nums
       LEFT OUTER JOIN Sales.Orders AS O
         ON DATEADD(day, Nums.n - 1, '20060101') = O.orderdate
WHERE  Nums.n <= DATEDIFF(day, '20060101', '20081231') + 1
ORDER BY orderdate;

```

Вот урезанная версия результата, который генерирует данный запрос.

orderdate	orderid	custid	empid
2006-01-01 00:00:00.000	NULL	NULL	NULL
2006-01-02 00:00:00.000	NULL	NULL	NULL
2006-01-03 00:00:00.000	NULL	NULL	NULL
2006-01-04 00:00:00.000	NULL	NULL	NULL
2006-01-05 00:00:00.000	NULL	NULL	NULL
...			
2006-06-29 00:00:00.000	NULL	NULL	NULL
2006-06-30 00:00:00.000	NULL	NULL	NULL
2006-07-01 00:00:00.000	NULL	NULL	NULL
2006-07-02 00:00:00.000	NULL	NULL	NULL
2006-07-03 00:00:00.000	NULL	NULL	NULL
2006-07-04 00:00:00.000	10248	85	5
2006-07-05 00:00:00.000	10249	79	6
2006-07-06 00:00:00.000	NULL	NULL	NULL
2006-07-07 00:00:00.000	NULL	NULL	NULL
2006-07-08 00:00:00.000	10250	34	4
2006-07-08 00:00:00.000	10251	84	3
2006-07-09 00:00:00.000	10252	76	4
2006-07-10 00:00:00.000	10253	34	3
2006-07-11 00:00:00.000	10254	14	5
2006-07-12 00:00:00.000	10255	68	9
2006-07-13 00:00:00.000	NULL	NULL	NULL
2006-07-14 00:00:00.000	NULL	NULL	NULL
2006-07-15 00:00:00.000	10256	88	3
2006-07-16 00:00:00.000	10257	35	4
...			
2008-12-27 00:00:00.000	NULL	NULL	NULL
2008-12-28 00:00:00.000	NULL	NULL	NULL
2008-12-29 00:00:00.000	NULL	NULL	NULL
2008-12-30 00:00:00.000	NULL	NULL	NULL
2008-12-31 00:00:00.000	NULL	NULL	NULL

(строк обработано: 1446)

Те даты, которых нет с правой стороны соединения, будут содержать значения NULL во всех столбцах таблицы `Orders`.

## Фильтрация атрибутов с противоположной стороны внешнего соединения

При поиске логических ошибок в коде, содержащем внешние соединения, особое внимание следует уделять инструкции `WHERE`. Если предикат, который в ней указан, ссылается на атрибут из противоположной стороны соединения и при этом используется выражение вида <атрибут> <оператор> <значение>, это обычно свидетельствует о некорректности запроса. Дело в том, что атрибуты, взятые из противоположной стороны соединения, во внешних строках равны NULL, а выражение `NULL <оператор> <значение>` возвращает `UNKNOWN` (разве что указан оператор `IS NULL`, который ищет отметки NULL). Напомним, что инструкция `WHERE` отбрасывает значения `UNKNOWN`. В итоге такой предикат отфильтровывает все внешние строки, нивелируя действие внешнего соединения. Другими словами, внешнее соединение превращается во внутреннее. Получается, что программист допустил ошибку либо при выборе типа соединения, либо в предикате. Если вы не очень понимаете, о чем речь, следующий пример должен все прояснить.

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
ON C.custid = O.custid
WHERE O.orderdate >= '20070101';
```

Этот запрос выполняет левое внешнее соединение таблиц `Customers` и `Orders`. Перед применением фильтра `WHERE` оператор `JOIN` возвращает внутренние и внешние записи о клиентах; у первых имеются заказы, а у вторых — нет (у них атрибуты таблицы `Orders` равны NULL). Предикат `O.orderdate >= '20070101'` в инструкции `WHERE` возвращает `UNKNOWN` для всех внешних строк, поскольку их атрибут `O.orderdate` равен NULL. В итоге все внешние строки отбрасываются на этапе `WHERE` — в этом можно убедиться, взглянув на результат, представленный ниже.

custid	companyname	orderid	orderdate
19	Клиент RFNQC	10400	2007-01-01 00:00:00.000
65	Клиент NYUHS	10401	2007-01-01 00:00:00.000
20	Клиент THNDP	10402	2007-01-02 00:00:00.000
20	Клиент THNDP	10403	2007-01-03 00:00:00.000
49	Клиент CQRAA	10404	2007-01-03 00:00:00.000
...			
58	Клиент AHXHT	11073	2008-05-05 00:00:00.000
73	Клиент JMIKW	11074	2008-05-06 00:00:00.000
68	Клиент CCKOT	11075	2008-05-06 00:00:00.000
9	Клиент RTXGC	11076	2008-05-06 00:00:00.000
65	Клиент NYUHS	11077	2008-05-06 00:00:00.000

(строк обработано: 678)

Это означает, что в данном случае использование внешнего соединения не имеет смысла. Допущена ошибка либо при выборе разновидности оператора `JOIN`, либо при написании предиката `WHERE`.



## Множественные внешние соединения

Вспомните, как мы обсуждали одновременные операции в главе 2. Тогда вы узнали, что все выражения, находящиеся на одном логическом этапе обработки, выполняются одновременно. Однако это не относится к табличным операторам в инструкции FROM. Они обрабатываются последовательно слева направо. Изменение этого порядка может сказаться на результате.

С порядком обработки внешних соединений связаны некоторые интересные логические ошибки. Возьмем для примера ситуацию, которую мы рассматривали выше. Представьте, что вам нужно написать запрос, который выполняет внешнее соединение двух таблиц, а потом добавляет третью таблицу с помощью внутреннего соединения. Если во внутреннем соединении предикат ON сравнивает атрибут из третьей таблицы с атрибутом из противоположной стороны внешнего соединения, все внешние строки отбрасываются. Как вы помните, внешние строки хранят в атрибутах, взятых с противоположной стороны, отметки NULL; при сравнении этих отметок с любым другим значением возвращается UNKNOWN. В итоге такие строки отфильтровываются предикатом ON. Проще говоря, этот предикат нивелирует действие внешнего соединения, превращая его, по сути, во внутреннее. Взгляните на следующий запрос.

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
     LEFT OUTER JOIN Sales.Orders AS O
       ON C.custid = O.custid
     JOIN Sales.OrderDetails AS OD
       ON O.orderid = OD.orderid;
```

Первым выполняется внешнее соединение, которое возвращает список клиентов (с заказами и без). Внешние строки, представляющие клиентов без заказов, содержат отметки NULL во всех атрибутах таблицы **Orders**. Дальше идет внутреннее соединение, которое с помощью предиката `O.orderid = OD.orderid` сопоставляет записи, полученные на предыдущем этапе, и строки из таблицы **OrderDetails**; однако в строках, представляющих клиентов без заказов, атрибут `O.orderid` равен NULL. Таким образом, эти строки отбрасываются, поскольку предикат возвращает UNKNOWN. В результирующий набор (его сокращенная версия показана ниже) не вошли записи с идентификаторами 22 и 57 — это те два клиента, которые не размещали заказов.

custid	orderid	productid	qty
85	10248	11	12
85	10248	42	10
85	10248	72	5
79	10249	14	9
79	10249	51	40
...			
65	11077	64	2
65	11077	66	1
65	11077	73	2
65	11077	75	4
65	11077	77	2

(строка обработано: 2155)

Внешние строки отбрасываются всегда, если за внешним соединением (левым, правым или полным) следует внутреннее при условии, что со значениями с правой стороны сравниваются отметки NULL.

Есть несколько способов решить эту проблему. Чтобы вернуть клиентов, у которых нет заказов, можно, например, вместо внутреннего соединения использовать левое внешнее.

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
  LEFT OUTER JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid;
```

Такой подход позволит сохранить внешние строки, сгенерированные в первом соединении. Ниже представлен результат (в сокращенном виде).

custid	orderid	productid	qty
85	10248	11	12
85	10248	42	10
85	10248	72	5
79	10249	14	9
79	10249	51	40
...			
65	11077	60	2
65	11077	64	2
65	11077	66	1
65	11077	73	2
65	11077	75	4
65	11077	77	2
22	NULL	NULL	NULL
57	NULL	NULL	NULL

(строк обработано: 2157)

Второй вариант: выполнить внутреннее соединение таблиц **Orders** и **OrderDetails**, и затем добавить к полученному результату таблицу **Customers**, используя правое внешнее соединение.

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON O.orderid = OD.orderid
  RIGHT OUTER JOIN Sales.Customers AS C
    ON O.custid = C.custid;
```

В таком случае внешние строки появятся на этапе последнего соединения и поэтому не будут отфильтрованы.

Третий вариант заключается в использовании скобок. С их помощью внутреннее соединение таблиц **Orders** и **OrderDetails** можно превратить в логически независимый этап. Это позволит выполнить левое внешнее соединение между таблицей **Customers** и результатом вышеупомянутого внутреннего соединения. Запрос будет выглядеть следующим образом.

```
SELECT C.custid, O.orderid, OD.productid, OD.qty
FROM Sales.Customers AS C
  LEFT OUTER JOIN
    (Sales.Orders AS O
      JOIN Sales.OrderDetails AS OD
        ON O.orderid = OD.orderid)
  ON C.custid = O.custid;
```

## Использование внешних соединений в сочетании с функцией COUNT

Еще одна распространенная логическая ошибка связана с совместным использованием внешних соединений и функции COUNT. Когда вы группируете результат внешнего соединения и агрегируете его с помощью выражения COUNT (\*), SQL Server обрабатывает как внешние, так и внутренние строки, независимо от их содержания. Обычно внешние строки не должны учитываться при подсчете количества элементов в результирующем наборе. Например, следующий запрос должен вернуть число заказов, сделанных каждым клиентом.

```
SELECT C.custid, COUNT(*) AS numorders
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON C.custid = O.custid
GROUP BY C.custid;
```

Однако агрегатная функция COUNT (\*) подсчитает все строки, независимо от их содержимого. Поэтому клиенты, которые не размещали заказов (под номерами 22 и 57), представлены в результирующем наборе внешними строками. Как можно видеть ниже, у каждой из этих двух записей количество заказов равно 1, хотя на самом деле там должно быть значение 0.

custid	numorders
1	6
2	4
3	7
4	13
5	18
...	
22	1
...	
57	1
...	
87	15
88	9
89	14
90	7
91	7

(строк обработано: 91)

Функция COUNT (\*) не может определить, представляет ли строка заказ. Чтобы решить эту проблему, вместо COUNT (\*) нужно использовать выражение

COUNT(<столбец>), указывая столбец из противоположной стороны соединения. В результате внешние строки будут игнорироваться, потому что в данном столбце они содержат значения NULL. При этом нужно использовать атрибут, который во внешних строках всегда равен NULL, например столбец первичного ключа `orderid`.

```
SELECT C.custid, COUNT(O.orderid) AS numorders
FROM Sales.Customers AS C
     LEFT OUTER JOIN Sales.Orders AS O
       ON C.custid = O.custid
GROUP BY C.custid;
```

Ниже представлена сокращенная версия конечного результата. Как видите, количество заказов у клиентов под номерами 22 и 57 равно 0.

custid	numorders
-----	-----
1	6
2	4
3	7
4	13
5	18
...	
22	0
...	
57	0
...	
87	15
88	9
89	14
90	7
91	7

(строк обработано: 91)

## В заключение

В этой главе вы познакомились с табличным оператором JOIN. Были рассмотрены логические этапы обработки, из которых состоят три основных типа соединений — перекрестное, внутреннее и внешнее. Вам также были представлены более сложные разновидности этого оператора, включая составные, множественные и не-эквисоединения. Завершает главу раздел, посвященный разным неочевидным аспектам внешних соединений; он здесь размещен на правах факультативного материала. Чтобы закрепить приобретенные знания, вы можете попрактиковаться с помощью упражнений, представленных ниже.

## Упражнения

Здесь собраны упражнения, которые помогут вам лучше понять материал, рассмотренный в этой главе. Все запросы выполняются к объектам БД TSQ12012.

**1-1**

Напишите запрос, который генерирует по пять копий каждой строки из таблицы `HR.Employees`.

Используются таблицы `HR.Employees` и `dbo.Nums`.

Ожидаемый результат:

empid	firstname	lastname	n
1	Сара	Дэвис	1
2	Дон	Функ	1
3	Джуди	Лью	1
4	Иаиль	Пелед	1
5	Свен	Бак	1
6	Пол	Суурс	1
7	Рассел	Кинг	1
8	Мария	Камерон	1
9	Зоя	Долгопятова	1
1	Сара	Дэвис	2
2	Дон	Функ	2
3	Джуди	Лью	2
4	Иаиль	Пелед	2
5	Свен	Бак	2
6	Пол	Суурс	2
7	Рассел	Кинг	2
8	Мария	Камерон	2
9	Зоя	Долгопятова	2
1	Сара	Дэвис	3
2	Дон	Функ	3
3	Джуди	Лью	3
4	Иаиль	Пелед	3
5	Свен	Бак	3
6	Пол	Суурс	3
7	Рассел	Кинг	3
8	Мария	Камерон	3
9	Зоя	Долгопятова	3
1	Сара	Дэвис	4
2	Дон	Функ	4
3	Джуди	Лью	4
4	Иаиль	Пелед	4
5	Свен	Бак	4
6	Пол	Суурс	4
7	Рассел	Кинг	4
8	Мария	Камерон	4
9	Зоя	Долгопятова	4
1	Сара	Дэвис	5
2	Дон	Функ	5
3	Джуди	Лью	5
4	Иаиль	Пелед	5
5	Свен	Бак	5
6	Пол	Суурс	5
7	Рассел	Кинг	5
8	Мария	Камерон	5
9	Зоя	Долгопятова	5

(строка обработано: 45)

## 1-2 (углубленное, по желанию)

Напишите запрос, который возвращает по одной строке для каждого сотрудника и дня в диапазоне с 12 по 16 июня 2009 г.

Используются таблицы `HR.Employees` и `dbo.Nums`.

Ожидаемый результат:

empid	dt
1	2009-06-12 00:00:00.000
1	2009-06-13 00:00:00.000
1	2009-06-14 00:00:00.000
1	2009-06-15 00:00:00.000
1	2009-06-16 00:00:00.000
2	2009-06-12 00:00:00.000
2	2009-06-13 00:00:00.000
2	2009-06-14 00:00:00.000
2	2009-06-15 00:00:00.000
2	2009-06-16 00:00:00.000
3	2009-06-12 00:00:00.000
3	2009-06-13 00:00:00.000
3	2009-06-14 00:00:00.000
3	2009-06-15 00:00:00.000
3	2009-06-16 00:00:00.000
4	2009-06-12 00:00:00.000
4	2009-06-13 00:00:00.000
4	2009-06-14 00:00:00.000
4	2009-06-15 00:00:00.000
4	2009-06-16 00:00:00.000
5	2009-06-12 00:00:00.000
5	2009-06-13 00:00:00.000
5	2009-06-14 00:00:00.000
5	2009-06-15 00:00:00.000
5	2009-06-16 00:00:00.000
6	2009-06-12 00:00:00.000
6	2009-06-13 00:00:00.000
6	2009-06-14 00:00:00.000
6	2009-06-15 00:00:00.000
6	2009-06-16 00:00:00.000
7	2009-06-12 00:00:00.000
7	2009-06-13 00:00:00.000
7	2009-06-14 00:00:00.000
7	2009-06-15 00:00:00.000
7	2009-06-16 00:00:00.000
8	2009-06-12 00:00:00.000
8	2009-06-13 00:00:00.000
8	2009-06-14 00:00:00.000
8	2009-06-15 00:00:00.000
8	2009-06-16 00:00:00.000
9	2009-06-12 00:00:00.000
9	2009-06-13 00:00:00.000
9	2009-06-14 00:00:00.000
9	2009-06-15 00:00:00.000
9	2009-06-16 00:00:00.000

(строк обработано: 45)

## 2

Запрос должен отобразить всех клиентов из США и вернуть для каждого из них количество сделанных заказов и общее число заказанных товаров.

Используются таблицы `Sales.Customers`, `Sales.Orders` и `Sales.OrderDetails`.

Ожидаемый результат:

custid	numorders	totalqty
-----	-----	-----
32	11	345
36	5	122
43	2	20
45	4	181
48	8	134
55	10	603
65	18	1383
71	31	4958
75	9	327
77	4	46
78	3	59
82	3	89
89	14	1063

(строк обработано: 13)

## 3

Нужно получить список клиентов с их заказами. В результат должны попасть даже те клиенты, которые ничего не заказывали.

Используются таблицы `Sales.Customers` и `Sales.Orders`.

Ожидаемый результат (в сокращенном виде):

custid	companyname	orderid	orderdate
-----	-----	-----	-----
85	Клиент ENQZT	10248	2006-07-04 00:00:00.000
79	Клиент FAPSM	10249	2006-07-05 00:00:00.000
34	Клиент IBVRG	10250	2006-07-08 00:00:00.000
84	Клиент NRCSK	10251	2006-07-08 00:00:00.000
...			
73	Клиент JMIKW	11074	2008-05-06 00:00:00.000
68	Клиент CCKOT	11075	2008-05-06 00:00:00.000
9	Клиент RTXGC	11076	2008-05-06 00:00:00.000
65	Клиент NYUHS	11077	2008-05-06 00:00:00.000
22	Клиент DTD MN	NULL	NULL
57	Клиент WVAXS	NULL	NULL

(строк обработано: 832)

## 4

Запрос должен вернуть всех клиентов, которые не делали заказов.

Используются таблицы `Sales.Customers` и `Sales.Orders`.

Ожидаемый результат:

custid	companyname
22	Клиент DTDMN
57	Клиент WVAXS

(строк обработано: 2)

5

Получите список клиентов, которые размещали заказы 12 февраля 2007 г. В результат должны войти столбцы из таблицы **Sales.Orders**.

Используются таблицы **Sales.Customers** и **Sales.Orders**.

Ожидаемый результат:

custid	companyname	orderid	orderdate
66	Клиент LHANT	10443	2007-02-12 00:00:00.000
5	Клиент HGVLZ	10444	2007-02-12 00:00:00.000

(строк обработано: 2)

6 (углубленное, по желанию)

Получите список клиентов, которые размещали заказы 12 февраля 2007 г. В результат должны войти столбцы из таблицы **Sales.Orders** и клиенты, у которых не было заказов в этот день.

Используются таблицы **Sales.Customers** и **Sales.Orders**.

Ожидаемый результат (в сокращенном виде):

custid	companyname	orderid	orderdate
72	Клиент AHPOP	NULL	NULL
58	Клиент AHXHT	NULL	NULL
25	Клиент AZJED	NULL	NULL
18	Клиент BSVAR	NULL	NULL
91	Клиент CCFIZ	NULL	NULL
...			
33	Клиент FVXPQ	NULL	NULL
53	Клиент GCJSG	NULL	NULL
39	Клиент GLLAG	NULL	NULL
16	Клиент GYBBY	NULL	NULL
4	Клиент HFBZG	NULL	NULL
5	Клиент HGVLZ	10444	2007-02-12 00:00:00.000
42	Клиент IAIJK	NULL	NULL
34	Клиент IBVRG	NULL	NULL
63	Клиент IRRVL	NULL	NULL
73	Клиент JMIKW	NULL	NULL
15	Клиент JUWXX	NULL	NULL
...			



```

21      Клиент KIDPX  NULL      NULL
30      Клиент KSLQF  NULL      NULL
55      Клиент KZQZT  NULL      NULL
71      Клиент LCOUJ  NULL      NULL
77      Клиент LCYBZ  NULL      NULL
66      Клиент LHANT  10443     2007-02-12 00:00:00.000
38      Клиент LJUCA  NULL      NULL
59      Клиент LOLJO  NULL      NULL
36      Клиент LVJSO  NULL      NULL
64      Клиент LWGMD  NULL      NULL
29      Клиент MDLWA  NULL      NULL
...
(строк обработано: 91)

```

## 7 (углубленное, по желанию)

Получите список всех клиентов. Результат должен содержать столбец со значениями «Да/Нет», которые определяются тем, размещал ли клиент заказ 12 февраля 2007 г.

Используются таблицы **Sales.Customers** и **Sales.Orders**.

Ожидаемый результат (в сокращенном виде):

custid	companyname	HasOrderOn20070212
1	Клиент NRZBB	Нет
2	Клиент MLTDN	Нет
3	Клиент KBUDE	Нет
4	Клиент HFBZG	Нет
5	Клиент HGVLZ	Да
6	Клиент XHXJV	Нет
7	Клиент QXVLA	Нет
8	Клиент QUHWH	Нет
9	Клиент RTXGC	Нет
10	Клиент EEALV	Нет

## Решения

Здесь приводятся решения для упражнений, представленных в этой главе.

### 1-1

Создание нескольких копий одних и тех же строк можно выполнить за счет базового приема, в основе которого лежит перекрестное соединение. Чтобы получить пять копий, вам надо иметь вспомогательную таблицу с пятью строками; на самом деле строк может быть и больше, но после соединения их нужно будет отфильтровать с помощью инструкции **WHERE**. Для этого прекрасно подойдет таблица **Nums**. Объедините ее с таблицей **Employees**, используя перекрестное соединение, затем оставьте столько строк, сколько требуется согласно условию (в нашем случае пять). Ниже представлено решение.

```
SELECT E.empid, E.firstname, E.lastname, N.n
FROM HR.Employees AS E
      CROSS JOIN dbo.Nums AS N
WHERE N.n <= 5
ORDER BY n, empid;
```

## 1-2

Это упражнение дополняет предыдущее. Вместо создания требуемого числа копий каждой записи о клиентах вас попросили сгенерировать копию каждого дня в определенном диапазоне дат. Для этого нужно вычислить количество дней в диапазоне, используя функцию `DATEDIFF`, и затем сослаться на полученный результат в инструкции `WHERE` — так же, как мы ранее ссылались на константу. Чтобы получить все необходимые дни, добавьте к начальной дате диапазона выражение `n - 1`. Вот как выглядит решение.

```
SELECT E.empid,
      DATEADD(day, D.n - 1, '20090612') AS dt
FROM HR.Employees AS E
      CROSS JOIN dbo.Nums AS D
WHERE D.n <= DATEDIFF(day, '20090612', '20090616') + 1
ORDER BY empid, dt;
```

Функция `DATEDIFF` возвращает 4, потому что между 12 и 16 июня 2009 г. разница в четыре дня. Добавьте к этому значению 1 — и вы получите 5 дней в диапазоне. В итоге инструкция `WHERE` отбирает пять строк из таблицы `Nums` с учетом того, что `n` меньше или равно 5. Добавляя `n - 1` дней к 12 июня 2009 г., вы получите все даты в нужном вам диапазоне.

## 2

В этом упражнении вам нужно было соединить три таблицы: `Customers`, `Orders` и `OrderDetails`. С помощью инструкции `WHERE` запрос должен отбирать только тех клиентов, которые живут в США. Согласно условию вам нужно вернуть результат выполнения агрегатной функции для каждого клиента, поэтому строки необходимо группировать по соответствующим идентификаторам. Чтобы получить правильное количество заказов для каждой записи, придется решить довольно замысловатую проблему. Соединение таблиц `Orders` и `OrderDetails` возвращает записи не для целых заказов, а для их составляющих, поэтому при использовании выражения `COUNT(*)` в инструкции `SELECT` вы, по сути, получите количество товаров, заказанных клиентами, но не количество самих заказов. Чтобы произвести корректный подсчет, каждый заказ нужно учитывать только один раз. Для этого вместо выражения `COUNT(*)` можно указать `COUNT(DISTINCT O.orderid)`. Это не вызовет проблем с вычислением общего количества товара, поскольку данное количество относится к отдельным составляющим заказа, а не к заказу целиком. Ниже представлено решение.

```
SELECT C.custid, COUNT(DISTINCT O.orderid) AS numorders, SUM(OD.qty) AS
totalqty
FROM Sales.Customers AS C
      JOIN Sales.Orders AS O
```

```
ON O.custid = C.custid
JOIN Sales.OrderDetails AS OD
  OD.orderid = O.orderid
WHERE C.country = N'США'
GROUP BY C.custid;
```

### 3

Чтобы в результирующий набор попали клиенты как с заказами, так и без них, вы должны использовать внешнее соединение наподобие следующего:

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON O.custid = C.custid;
```

Этот запрос возвращает 832 строки. Внутреннее соединение вернуло бы только 830 строк без записей о клиентах с идентификаторами 22 и 57, которые не размещали заказов.

### 4

Данное упражнение дополняет предыдущее. Чтобы получить список клиентов, которые не делали заказов, вам нужно добавить в свой запрос инструкцию `WHERE`; она будет отбирать только внешние строки — записи, которые представляют клиентов без заказов. В атрибутах, взятых с противоположной стороны соединения (таблица `Orders`), внешние строки содержат значения `NULL`. Однако отметки `NULL` могут быть как заглушками для внешних строк, так и реальными значениями из таблицы. Чтобы их отличать, рекомендуется ссылаться в запросе на атрибут из первичного ключа или соединения; вы также можете выбрать столбец, который не поддерживает отметки `NULL`. В решении, представленном ниже, инструкция `WHERE` содержит ссылку на первичный ключ из таблицы `Orders`.

```
SELECT C.custid, C.companyname
FROM Sales.Customers AS C
  LEFT OUTER JOIN Sales.Orders AS O
    ON O.custid = C.custid
WHERE O.orderid IS NULL;
```

В результате мы получим всего две записи. Они представляют клиентов с идентификаторами 22 и 57, которые не делали заказов.

### 5

Чтобы выполнить это упражнение, нужно написать запрос, который выполняет внутреннее соединение таблиц `Customers` и `Orders`, отбирая строки с датой заказа «12 февраля 2007 г.».

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
  JOIN Sales.Orders AS O
    ON O.custid = C.custid
```

```
WHERE O.orderdate = '20070212';
```

Инструкция WHERE отфильтровывает клиентов, которые не размещали заказов 12 февраля 2007 г., что и требовалось в условии.

## 6

Это упражнение основано на предыдущем. Здесь, во-первых, нужно использовать внешнее соединение, ведь вам необходимо вернуть клиентов, которые не отвечают заданным критериям. Во-вторых, фильтрация по дате заказа должна происходить в предикате ON, а не WHERE. Не забывайте, что фильтр WHERE находится в самом конце и применяется уже после включения внешних строк. Вам нужно сопоставлять клиентов только с теми заказами, которые были размещены 12 февраля 2007 г. Также следует получить список клиентов, которые не делали заказов в этот день; следовательно, фильтрация по дате должна просто определять соответствие записей, но не формировать окончательный список клиентов. Таким образом, предикат ON должен сопоставить записи из таблиц Customers и Orders, учитывая идентификаторы клиентов и дату заказов (12 февраля 2007 г.). Решение представлено ниже.

```
SELECT C.custid, C.companyname, O.orderid, O.orderdate
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
ON O.custid = C.custid
AND O.orderdate = '20070212';
```

## 7

Это упражнение также является расширением предыдущего. Здесь нужно возвращать значения «Да/Нет», которые сообщают о наличии или отсутствии подходящих заказов. Как вы помните, строки, которые не соответствуют условию соединения, являются внешними и содержат отметки NULL в атрибутах, взятых с противоположной стороны. Чтобы проверить, является ли строка внешней, воспользуйтесь выражением CASE; при положительном ответе вернется Да, а при отрицательном — Нет. Так как теоретически на одного клиента может приходиться несколько соответствий, укажите в списке SELECT инструкцию DISTINCT. Так вы получите только по одной строке для каждого клиента. Вот это решение.

```
SELECT DISTINCT C.custid, C.companyname,
CASE WHEN O.orderid IS NOT NULL THEN 'Да' ELSE 'Нет' END AS
[HasOrderOn20070212]
FROM Sales.Customers AS C
LEFT OUTER JOIN Sales.Orders AS O
ON O.custid = C.custid
AND O.orderdate = '20070212';
```

## Глава 4

# ВЛОЖЕННЫЕ ЗАПРОСЫ

Язык SQL поддерживает написание вложенных запросов. Запрос, который возвращает конечный результат, называют **внешним**. Внутри себя он может использовать результат выполнения других запросов. **Внутренние запросы** вызываются во время выполнения кода и являются аналогами выражений, основанных на переменных или константах. В отличие от выражений их результат может изменяться в зависимости от содержимого таблицы. Использование вложенных запросов избавляет от необходимости хранить промежуточные результаты в отдельных переменных.

Вложенные запросы могут быть либо автономными, либо коррелирующими. Первые, в отличие от вторых, никак не связаны со своим внешним кодом. Результат вложенного запроса может состоять из одного или нескольких значений или же вовсе являться таблицей.

В этой главе в основном рассматриваются варианты с одиночными (скалярными) и множественными значениями. Вложенные запросы, возвращающие целые таблицы, мы обсудим в главе 5.

Первым делом нам предстоит познакомиться с автономными вложенными запросами, уделяя особое внимание разнице между скалярными и множественными результатами. При рассмотрении коррелирующих запросов я буду исходить из того, что вы уже усвоили этот материал.

С помощью упражнений, размещенных в конце главы, вам удастся применить на практике полученные знания.

## Автономные вложенные запросы

Любой вложенный запрос вызывается извне, однако автономные не зависят от внешнего кода. Их очень удобно отлаживать, вы всегда можете выполнить их отдельно и посмотреть, делают ли они то, что нужно. Сначала выполняется вложенный запрос, а потом внешний код использует полученный результат в своих вычислениях. Конкретные примеры того, как это происходит, рассматриваются в следующих подразделах.

### Примеры со скалярным результатом

Скалярный вложенный запрос всегда возвращает одно значение независимо от того, автономный он или коррелирующий. Он может находиться в любом месте

внешнего кода, где допустимо использование скалярных выражений (например, в инструкциях WHERE или SELECT).

Представьте, что вам нужно извлечь из таблицы **Orders** информацию о заказе с наибольшим значением идентификатора. Сначала это значение необходимо сохранить в переменной, а потом использовать его внутри фильтра, чтобы найти нужный заказ. Этот подход продемонстрирован в следующем примере.

```
USE TSQL2012;

DECLARE @maxid AS INT = (SELECT MAX(orderid)
                        FROM Sales.Orders);

SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderid = @maxid;
```

Ниже представлен результат.

orderid	orderdate	empid	custid
11077	2008-05-06	00:00:00.000	1 65

Вместо ссылки на переменную можно указать автономный вложенный запрос, возвращающий скалярное значение, то есть самый большой имеющийся идентификатор. Так вы сможете превратить двухэтапный процесс в единственный запрос.

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderid = (SELECT MAX(O.orderid)
                FROM Sales.Orders AS O);
```

Скалярный вложенный запрос должен возвращать не больше одного значения, в противном случае он может завершиться ошибкой во время выполнения кода. Следующий код на первый взгляд является корректным.

```
SELECT orderid
FROM Sales.Orders
WHERE empid =
    (SELECT E.empid
     FROM HR.Employees AS E
     WHERE E.lastname LIKE N'B%');
```

Задача этого кода — получить идентификаторы заказов, размещенных клиентами, фамилии которых начинаются на букву «Б». Вложенный запрос возвращает идентификаторы клиентов, чьи фамилии удовлетворяют вышеупомянутому условию, а затем во внешнем соединении этот список используется для получения номеров заказов. Поскольку оператор равенства принимает с обеих сторон одиначные значения, вложенный запрос считается скалярным, хотя теоретически он может вернуть сразу несколько результатов. В этом и кроется ошибка — оператор равенства не сможет корректно обработать больше одного значения.

Нам просто повезло, что в таблице **Employees** есть данные только об одном сотруднике, имя которого начинается на букву «Б» (Свен Бак с идентификатором 5). Ниже показан результат, который вернет наш запрос (в сокращенном виде).

```
orderid
-----
10248
10254
10269
10297
10320
...
10874
10899
10922
10954
11043
```

(строк обработано: 42)

Конечно, если вложенный запрос вернет больше одного значения, этот код завершится ошибкой. К примеру, попробуйте изменить условие так, чтобы отбирались сотрудники с фамилией на букву «Д».

В нашей таблице есть записи о двух таких сотрудниках — это Сара Дэвис и Зоя Долгопятова. Поэтому запрос будет прерван и завершится следующей ошибкой.

Сообщение 512, уровень 16, состояние 1, строка 1  
Вложенный запрос вернул больше одного значения. Это запрещено, когда вложенный запрос следует после =, !=, <, <=, >, >= или используется в качестве выражения.

Если у скалярного вложенного запроса нет результата, он возвращает NULL. Как вы помните, сравнение NULL с любым значением дает UNKNOWN, в результате чего запись не проходит фильтр и отбрасывается. Например, в таблице `Employees` нет данных о сотрудниках с фамилиями на букву «А», поэтому следующий запрос вернет пустой набор.

```
SELECT orderid
FROM Sales.Orders
WHERE empid =
  (SELECT E.empid
   FROM HR.Employees AS E
   WHERE E.lastname LIKE N'A%');
```

## Примеры с множественными значениями

Существуют вложенные запросы (не важно, автономные или нет), которые возвращают несколько значений в виде единого столбца. Для работы с ними потребуются специальные предикаты, например инструкция `IN`.



### ПРИМЕЧАНИЕ

Есть и другие предикаты, поддерживающие вложенные запросы с множественными значениями (`SOME`, `ANY` и `ALL`). Так как они редко используются на практике, мы не будем рассматривать их в этой книге.

Предикат `IN` имеет следующий синтаксис:

<скалярное\_выражение> `IN` (<вложенный запрос с множественными значениями>)

Предикат является истинным, если скалярное выражение равно любому из значений, возвращенных вложенным запросом. Вспомните последний пример, который мы обсуждали в предыдущем разделе, — получение номеров заказов, обработанных сотрудниками, чья фамилия начинается с определенной буквы. В этой ситуации следует использовать предикат `IN` и вложенный запрос, возвращающий множественные значения (вместо оператора равенства и скалярного вложенного запроса), так как условию может удовлетворять сразу несколько сотрудников. Например, следующий запрос возвращает идентификаторы заказов, обработанных сотрудниками, у которых фамилии начинаются на букву «Д».

```
SELECT orderid
FROM Sales.Orders
WHERE empid IN
    (SELECT E.empid
     FROM HR.Employees AS E
     WHERE E.lastname LIKE N'Д%');
```

Благодаря использованию предиката `IN` этот запрос остается корректным при любом количестве возвращаемых значений (в том числе при нулевом). Ниже показана часть полученного результата.

```
orderid
-----
10258
10270
10275
10285
10292
...
10978
11016
11017
11022
11058
```

(строк обработано: 166)

Вам наверняка интересно, почему для решения этой задачи я выбрал именно вложенный запрос, а не соединение. Предыдущий пример выглядел бы следующим образом.

```
SELECT O.orderid
FROM HR.Employees AS E
    JOIN Sales.Orders AS O
      ON E.empid = O.empid
WHERE E.lastname LIKE N'Д%'
```

Существует множество ситуаций, в которых можно использовать как вложенные запросы, так и соединения. Нельзя сказать однозначно, какой из этих подходов является более удачным. Иногда они интерпретируются совершенно одинаково,



иногда проявляются отличия в производительности. Я предпочитаю решать задачу самым очевидным способом и только потом, если меня не устраивает скорость работы, пытаюсь видоизменить свой запрос, например заменить вложенные запросы на соединения (и наоборот).

Рассмотрим еще один пример вложенного запроса с множественными значениями. Представьте, что вам нужно получить список заказов, размещенных клиентами из США. Ваш запрос должен обращаться к таблице **Orders** и возвращать заказы, сделанные людьми, которые входят в множество клиентов, проживающих в США. Часть кода можно реализовать в виде автономного вложенного запроса с множественными значениями. Ниже представлено полноценное решение.

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders
WHERE custid IN
  (SELECT C.custid
   FROM Sales.Customers AS C
   WHERE C.country = N'США');
```

Результат (точнее, его сокращенная форма) будет выглядеть следующим образом.

custid	orderid	orderdate	empid
65	10262	2006-07-22 00:00:00.000	8
89	10269	2006-07-31 00:00:00.000	5
75	10271	2006-08-01 00:00:00.000	6
65	10272	2006-08-02 00:00:00.000	6
65	10294	2006-08-30 00:00:00.000	4
...			
32	11040	2008-04-22 00:00:00.000	4
32	11061	2008-04-30 00:00:00.000	4
71	11064	2008-05-01 00:00:00.000	1
89	11066	2008-05-01 00:00:00.000	7
65	11077	2008-05-06 00:00:00.000	1

(строк обработано: 122)

Инструкцию **IN**, как и любой другой предикат, можно использовать в связке с оператором отрицания **NOT**. К примеру, следующий запрос возвращает записи о клиентах, не разместивших ни одного заказа.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN
  (SELECT O.custid
   FROM Sales.Orders AS O);
```

При использовании вложенных запросов рекомендуется исключать отметки **NULL**. Здесь я этого не делаю, чтобы не усложнять код, и позже в разделе «Проблемы с отметками **NULL**» текущей главы объясню, почему.

Автономный вложенный запрос с множественными значениями возвращает идентификаторы всех клиентов, которые содержатся в таблице **Orders**. Внешний

код возвращает записи о клиентах из таблицы **Customers**, идентификаторы которых входят в набор значений, полученных из вложенного запроса, то есть список клиентов, которые не размещали заказов. Результат будет следующим.

custid	companyname
22	Клиент DTDMN
57	Клиент WVAXS

Вас, наверное, интересует, улучшит ли производительность инструкция **DISTINCT**, если ее добавить внутрь вложенного запроса, ведь в таблице **Orders** может находиться несколько записей с одним и тем же идентификатором клиента. Не стоит об этом беспокоиться, поскольку ядро БД умеет автоматически убирать дубликаты, даже если вы сами об этом не попросите.

В последнем примере, который приводится в текущем разделе, показано, как использовать сочетание из нескольких вложенных автономных запросов (как с одиночными, так и с множественными значениями). Прежде чем ознакомиться с сутью задачи, вы должны создать в БД **TSQL2012** таблицу под названием **dbo.Orders** и заполнить ее четными идентификаторами из таблицы **Sales.Orders**.

```
USE TSQL2012;
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
CREATE TABLE dbo.Orders(orderid INT NOT NULL CONSTRAINT PK_Orders PRIMARY
KEY);

INSERT INTO dbo.Orders(orderid)
SELECT orderid
FROM Sales.Orders
WHERE orderid % 2 = 0;
```

Если вам не понятно, как работает инструкция **INSERT**, не переживайте — я подробно опишу ее в главе 8.

Задача состоит в том, чтобы получить все пропущенные идентификаторы заказов, отбрасывая минимальное и максимальное значения. Это может оказаться довольно сложно, если не использовать никаких вспомогательных таблиц. В главе 3 мы уже успешно применяли таблицу **Nums**, которая, как вы помните, содержит последовательность целых чисел, начиная с 1 (без промежутков).

Чтобы получить из таблицы **Orders** все пропущенные идентификаторы, выполните запрос к таблице **Nums** и отберите только те значения, которые находятся в таблице **dbo.Orders** (исключая минимальное и максимальное) и не входят в множество значений **Orders**. Чтобы получить минимальное и максимальное значения идентификатора, можно использовать скалярный автономный вложенный запрос. Для получения списка всех существующих идентификаторов подойдет автономный вложенный запрос с множественными значениями.

```
SELECT n
FROM dbo.Nums
WHERE n BETWEEN (SELECT MIN(O.orderid) FROM dbo.Orders AS O)
AND (SELECT MAX(O.orderid) FROM dbo.Orders AS O)
AND n NOT IN (SELECT O.orderid FROM dbo.Orders AS O);
```

Код, наполнявший данными таблицу `dbo.Orders`, отфильтровал все четные идентификаторы, поэтому наш запрос возвращает нечетные значения в диапазоне между минимальным и максимальным идентификаторами, содержащимися в таблице `Orders`. Ниже представлен результат (в сокращенном виде).

```
n
-----
10249
10251
10253
10255
10257
...
11067
11069
11071
11073
11075
```

(строк обработано: 414)

Закончив с этим примером, можете удалить таблицу `dbo.Orders`, используя следующий код

```
DROP TABLE dbo.Orders;
```

## Коррелирующие вложенные запросы

Коррелирующие вложенные запросы ссылаются на атрибуты таблицы, указанной во внешнем коде, то есть вложенный запрос зависит от внешнего кода и не может работать самостоятельно. С логической точки зрения это выглядит так, как будто вложенный запрос запускается отдельно для каждой внешней строки. В качестве примера рассмотрим листинг 4.1, который возвращает идентификатор заказа с максимальным значением для каждого клиента.

### Листинг 4.1. Коррелирующий вложенный запрос

```
USE TSQL2012;
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders AS O1
WHERE orderid =
    (SELECT MAX(O2.orderid)
     FROM Sales.Orders AS O2
     WHERE O2.custid = O1.custid);
```

Внешний код обращается к экземпляру таблицы `Orders` под названием `O1`; он отбирает заказы, у которых идентификатор равен значению, полученному из вложенного запроса. Тот в свою очередь отфильтровывает заказы из второго экземпляра таблицы `Orders` под названием `O2`, возвращая для текущего клиента идентификатор заказа с максимальным значением; согласно условию внутренний и внешний идентификаторы клиентов должны быть равны. Проще говоря, вложенный запрос перебирает все строки внутри `O1` и возвращает максимальный номер заказа для текущего клиента.

Если идентификаторы заказов из таблицы **O1** и вложенного запроса совпадают, значит, мы имеем максимальный номер заказа для текущего клиента, в результате чего запрос возвращает строку из **O1**. Сокращенный результат приводится ниже.

custid	orderid	orderdate	empid
91	11044	2008-04-23 00:00:00.000	4
90	11005	2008-04-07 00:00:00.000	2
89	11066	2008-05-01 00:00:00.000	7
88	10935	2008-03-09 00:00:00.000	4
87	11025	2008-04-15 00:00:00.000	6
...			
5	10924	2008-03-04 00:00:00.000	3
4	11016	2008-04-10 00:00:00.000	9
3	10856	2008-01-28 00:00:00.000	3
2	10926	2008-03-04 00:00:00.000	4
1	11011	2008-04-09 00:00:00.000	3

(строк обработано: 89)

Коррелирующие вложенные запросы обычно выглядят сложнее, чем автономные. Я советую вам сосредоточиться на какой-то одной строке из внешней таблицы, в частности на том, как эта строка обрабатывается. Возьмем для примера заказ из таблицы **Orders** под номером 10248.

custid	orderid	orderdate	empid
85	10248	2006-07-04 00:00:00.000	5

Когда вложенный запрос будет выполняться для этой внешней строки, ссылка на атрибут **O1.custid** (корреляция) будет равна 85. Заменив ссылку данным значением, мы получим следующий код.

```
SELECT MAX(O2.orderid)
FROM Sales.Orders AS O2
WHERE O2.custid = 85;
```

Этот запрос возвращает идентификатор заказа со значением 10 739. В нашей строке идентификатор равен 10 248. Совпадения не обнаружено, поэтому внешняя строка отбрасывается. То же самое будет с остальными строками, которые имеют идентификатор 10 248. Только когда попадется максимальный номер заказа для текущего клиента, мы получим совпадение. Такой подход должен помочь вам лучше понять принцип работы коррелирующих запросов.

Коррелирующие вложенные запросы, в отличие от автономных, зависят от внешнего кода. Их сложнее отлаживать, поскольку нельзя запускать отдельно. Например, попытавшись выполнить вложенный запрос из листинга 4.1, вы получите следующую ошибку.

```
Сообщение 4104, уровень 16, состояние 1, строка 3
Не удалось привязать составной идентификатор "O1.custid".
```

Данная ошибка сигнализирует о том, что идентификатор **O1.custid** не может быть привязан к объекту, поскольку псевдоним **O1** не был определен внутри запроса. Контекст этой ссылки находится во внешнем коде. Чтобы отлаживать коррелирующие

вложенные запросы, вместо корреляций нужно подставлять константы; убедившись, что код работает корректно, вы сможете произвести обратную замену.

Давайте рассмотрим еще один пример. Представьте, что вам нужно сделать запрос к представлению `Sales.OrderValues`, чтобы получить удельный вес каждого заказа в общей стоимости всех размещенных клиентами. В главе 7 я покажу, как решить эту задачу с помощью оконных функций; сейчас мы воспользуемся вложенными запросами. Старайтесь находить для каждой проблемы несколько решений, потому что разница в подходах обычно отражается на сложности и производительности вашего кода.

Мы можем выполнить к представлению `OrderValues` внешний запрос (используя псевдоним `O1`); в списке `SELECT` поделим текущее значение на результат коррелирующего вложенного запроса, который возвращает суммарную стоимость для текущего клиента, вычисленную с помощью второго экземпляра `OrderValues` (с псевдонимом `O2`). Вот как выглядит весь код целиком.

```
SELECT orderid, custid, val,
       CAST(100. * val / (SELECT SUM(O2.val)
                        FROM Sales.OrderValues AS O2
                        WHERE O2.custid = O1.custid)
           AS NUMERIC(5,2)) AS pct
FROM Sales.OrderValues AS O1
ORDER BY custid, orderid;
```

С помощью функции `CAST` значение, возвращаемое выражением, приводится к типу `NUMERIC` (трехзначному числу с двумя знаками после запятой).

Запрос возвращает следующий результат.

orderid	custid	val	pct
10643	1	814.50	19.06
10692	1	878.00	20.55
10702	1	330.00	7.72
10835	1	845.80	19.79
10952	1	471.20	11.03
11011	1	933.50	21.85
10308	2	88.80	6.33
10625	2	479.75	34.20
10759	2	320.00	22.81
10926	2	514.40	36.67
...			

(строк обработано: 830)

## Предикат EXISTS

Язык T-SQL поддерживает предикат под названием `EXISTS`, который возвращает `TRUE` или `FALSE` в зависимости от того, генерирует ли переданный ему вложенный запрос какие-либо строки. Например, следующий код возвращает всех испанских клиентов, которые размещали заказы.

```
SELECT companyname, custid
FROM Sales.Customers AS C
WHERE country = N'Испания'
AND EXISTS
```

```
(SELECT * FROM Sales.Orders AS O
WHERE O.custid = C.custid);
```

Внешний запрос к таблице **Customers** отбирает клиентов из Испании, для которых предикат **EXISTS** возвращает **TRUE**, а это происходит только в случае, когда у текущего клиента есть хотя бы один заказ в таблице **Orders**.

Одно из преимуществ предиката **EXISTS** заключается в том, что он позволяет формировать запросы в виде фраз на английском языке. Например, вышеприведенный запрос в переводе с английского дословно звучит так: «выбрать из таблицы **Customers** атрибуты с названиями компаний и идентификаторами клиентов, где страна — Испания, и если в таблице **Orders** существует как минимум один заказ с тем же идентификатором клиента».

Результат, возвращаемый этим запросом, представлен ниже.

companyname	custid
-----	-----
Клиент QHWH	8
Клиент MDLWA	29
Клиент KSLQF	30
Клиент SIUIN	69

Как и другие предикаты, инструкцию **EXISTS** можно использовать в сочетании с логическим оператором отрицания **NOT**. Например, следующий запрос возвращает испанских клиентов, которые не размещали заказов.

```
SELECT companyname, custid
FROM Sales.Customers AS C
WHERE country = N'Испания'
AND NOT EXISTS
  (SELECT * FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
```

Ниже показан результат.

companyname	custid
-----	-----
Клиент DTDMM	22

Несмотря на то что в этой книге основной акцент делается на логической обработке запросов, а не на их производительности, вам будет интересно узнать, что предикат **EXISTS** хорошо поддается оптимизации. Ядро Microsoft SQL Server знает, что не нужно обрабатывать все запрашиваемые строки — достаточно получить хотя бы один результат (или его отсутствие).

Это тот редкий случай, когда в использовании символа звездочки (\*) внутри инструкции **SELECT** нет ничего плохого. Предикат **EXISTS** следит исключительно за наличием подходящих строк, игнорируя атрибуты, указанные для выборки, словно инструкция **SELECT** является лишней. Ядро SQL Server знает об этом и в целях оптимизации пропускает обработку списка **SELECT** внутри вложенного запроса. Поэтому использование группового символа \* вместо перечисления констант не влияет на производительность. Справедливости ради нужно отметить, что на процесс поиска столбцов все же могут быть выделены незначительные ресурсы, но вряд ли вы это заметите. Я считаю, что запросы должны выглядеть естественно

кода можно смириться с незначительными накладными расходами, связанными с интерпретацией символа \*.

Следует упомянуть об еще одной интересной особенности инструкции EXISTS: в отличие от большинства других предикатов в языке T-SQL она использует двоичную логику. Действительно, не существует такой ситуации, в которой на вопрос «возвращает ли код строки?» нельзя было бы ответить однозначно.

## Примеры сложных вложенных запросов

В этом разделе рассматриваются более сложные аспекты вложенных запросов. Материал, который здесь приводится, не является обязательным к прочтению; вы можете вернуться к нему позже, когда усвоите основы.

### Возвращение предыдущих или следующих значений

Представьте, что вместе с каждой записью в таблице **Orders** вам нужно получить идентификатор предыдущего заказа. Само понятие «предыдущий» подразумевает наличие какой-то логической упорядоченности. Так как строки в таблице не имеют никакого порядка, вам необходимо составить выражение на языке T-SQL, которое будет его эмулировать. За основу можно взять принцип, согласно которому предыдущим является «максимальное значение, которое меньше текущего». Эту фразу легко выразить в виде коррелирующего вложенного запроса, как показано ниже.

```
SELECT orderid, orderdate, empid, custid,
       (SELECT MAX(O2.orderid)
        FROM Sales.Orders AS O2
        WHERE O2.orderid < O1.orderid) AS prevorderid
FROM Sales.Orders AS O1;
```

Этот запрос возвращает следующий результат (представленный в сокращенном виде).

orderid	orderdate	empid	custid	prevorderid
10248	2006-07-04 00:00:00.000	5	85	NULL
10249	2006-07-05 00:00:00.000	6	79	10248
10250	2006-07-08 00:00:00.000	4	34	10249
10251	2006-07-08 00:00:00.000	3	84	10250
10252	2006-07-09 00:00:00.000	4	76	10251
...				
11073	2008-05-05 00:00:00.000	2	58	11072
11074	2008-05-06 00:00:00.000	7	73	11073
11075	2008-05-06 00:00:00.000	8	68	11074
11076	2008-05-06 00:00:00.000	4	9	11075
11077	2008-05-06 00:00:00.000	1	65	11076

(строк обработано: 830)

Как видите, у первой записи нет «предыдущего заказа», поэтому ее атрибут `prevorderid` содержит значение `NULL`.

Точно так же можно сказать, что следующая запись — это «минимальное значение, которое больше текущего». Код на языке T-SQL, представленный ниже, возвращает идентификатор следующего заказа для каждой строки в таблице **Orders**.

```
SELECT orderid, orderdate, empid, custid,
       (SELECT MIN(O2.orderid)
        FROM Sales.Orders AS O2
        WHERE O2.orderid > O1.orderid) AS nextorderid
FROM Sales.Orders AS O1;
```

Результат (точнее, его сокращенный вариант) выглядит так.

orderid	orderdate	empid	custid	nextorderid
-----	-----	-----	-----	-----
10248	2006-07-04 00:00:00.000	5	85	10249
10249	2006-07-05 00:00:00.000	6	79	10250
10250	2006-07-08 00:00:00.000	4	34	10251
10251	2006-07-08 00:00:00.000	3	84	10252
10252	2006-07-09 00:00:00.000	4	76	10253
...				
11073	2008-05-05 00:00:00.000	2	58	11074
11074	2008-05-06 00:00:00.000	7	73	11075
11075	2008-05-06 00:00:00.000	8	68	11076
11076	2008-05-06 00:00:00.000	4	9	11077
11077	2008-05-06 00:00:00.000	1	65	NULL

(строк обработано: 830)

Обратите внимание: у последней записи нет «следующего заказа», поэтому ее атрибут `prevorderid` содержит значение `NULL`.

В SQL Server 2012 появились новые оконные функции, `LAG` и `LEAD`, которые позволяют возвращать элемент из предыдущей или следующей строки, исходя из определенного порядка сортировки. Подробнее о них мы поговорим в главе 7.

## Использование текущих агрегатов

**Текущими агрегатами** называют результат суммирования значений по времени с помощью агрегатных функций. Чтобы продемонстрировать процесс вычисления текущих агрегатов, воспользуемся представлением `Sales.OrderTotalsByYear`, которое возвращает общее количество заказов за определенный год. Выполним запрос к этому представлению, чтобы проанализировать его содержимое.

```
SELECT orderyear, qty
FROM Sales.OrderTotalsByYear;
```

Вы получите следующий результат.

orderyear	qty
-----	-----
2007	25489
2008	16247
2006	9581

Допустим, для каждой строки вам нужно получить год, годовое количество и общее текущее количество заказов (сумму заказов, сделанных до конца определенного



года). Итак, для самой ранней записи в представлении (2006 г.) общее текущее количество будет равно сумме заказов, сделанных за текущий год. Для второй записи нужно сложить количество заказов за текущий (2007 г.) и предыдущий годы и т. д.

Эту задачу можно решить следующим образом: извлечь из первого экземпляра представления (назовем его O1) год и количество для каждой записи и затем с помощью коррелирующего вложенного запроса ко второму экземпляру представления (O2) вычислить общее текущее количество. Вложенный запрос должен отбирать из O2 все записи, сделанные за текущий или предыдущие года (относительно O1), суммируя количество заказов, размещенных за этот период. Вот как выглядит решение.

```
SELECT orderyear, qty,
       (SELECT SUM(O2.qty)
        FROM Sales.OrderTotalsByYear AS O2
        WHERE O2.orderyear <= O1.orderyear) AS runqty
FROM Sales.OrderTotalsByYear AS O1
ORDER BY orderyear;
```

Результат будет следующим.

orderyear	qty	runqty
2006	9581	9581
2007	25489	35070
2008	16247	51317

Стоит отметить, что в SQL Server 2012 были улучшены возможности оконных агрегатных функций, в частности появился новый высокоэффективный инструмент для вычисления общих текущих значений. Но, как я уже упоминал, об этом мы поговорим в главе 7.

## Решение проблем

В этом разделе рассматриваются ситуации, в которых вложенные запросы могут вести себя не так, как вы того ожидаете, а еще приводятся оптимальные решения, которые позволяют избежать подобных логических ошибок.

### Отметки NULL

Как вы помните, в языке T-SQL используется троичная логика. В этом разделе я продемонстрирую проблемы, которые могут возникнуть с вложенными запросами и отметками NULL, если не брать во внимание третье возможное значение (UNKNOWN).

Давайте рассмотрим простой на первый взгляд запрос, который должен возвращать список клиентов, не разместивших ни одного заказа.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN(SELECT O.custid
                    FROM Sales.Orders AS O);
```

Учитывая те данные, которые хранятся в таблице **Orders**, этот запрос должен работать так, как вы ожидаете; в самом деле, он возвращает две строки с информацией о двух клиентах, которые не делали заказов.

custid	companyname
22	Клиент DTDMN
57	Клиент WVAXS

Теперь запустите следующий код, чтобы вставить в таблицу **Orders** новый заказ, идентификатор которого будет равен NULL.

```
INSERT INTO Sales.Orders
(custid, empid, orderdate, requireddate, shippeddate,
shipperid, freight, shipname, shipaddress, shipcity,
shipregion, shippostalcode, shipcountry)
VALUES (NULL, 1, '20090212', '20090212',
'20090212', 1, 123.00, N'abc', N'abc', N'abc',
N'abc', N'abc', N'abc');
```

Попробуем снова выполнить запрос, который должен возвращать клиентов без заказов.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN (SELECT O.custid
FROM Sales.Orders AS O);
```

На этот раз запрос вернет пустой набор. Вспоминая то, что вы уже успели прочитать об отметках NULL в главе 2, попробуйте объяснить появление такого результата. Подумайте, как теперь получить список клиентов с идентификаторами 22 и 57, а также попробуйте сформулировать правила, следуя которым, можно избежать подобных проблем (если исходить из того, что это действительно проблема).

Очевидно, виновником в данной ситуации является запись с неопределенным идентификатором, которая была добавлена в таблицу **Orders**; она возвращается вложенным запросом вместе с другими записями, чьи идентификаторы определены.

Давайте начнем с того участка кода, который ведет себя предсказуемо. Предикат **IN** возвращает **TRUE** для клиента, который делал заказы (например, под номером 85), потому что запись с таким идентификатором возвращается вложенным запросом. Оператор **NOT** инвертирует предикат **IN**; следовательно, **NOT TRUE** превращается в **FALSE** и клиент «отбрасывается» внешним кодом. Итак, если идентификатор клиента находится в таблице **Orders**, можно быть уверенным в том, что он делал заказы и поэтому не должен попасть в результирующий набор. Но, как вы вскоре убедитесь, нельзя однозначно утверждать, что клиент с идентификатором NULL не находится в таблице **Orders**.

Для клиентов, чьи идентификаторы не значатся в таблице **Orders** как определенные (например, под номером 22), предикат **IN** возвращает **UNKNOWN** (полноценное значение истинности, ничем не хуже **TRUE** или **FALSE**). Это вызвано тем, что для определенных значений предикат **IN** возвращает **FALSE**, а для отметок NULL — **UNKNOWN**. Выражение **FALSE OR UNKNOWN** тоже дает **UNKNOWN**. В качестве более реального примера возьмем выражение **22 NOT IN (1, 2, NULL)**. Его можно

записать как `NOT 22 IN (1, 2, NULL)` или `NOT (22 = 1 OR 22 = 2 OR 22 = NULL)`. Заменим все операторы равенства, находящиеся в скобках, их значениями истинности; получится выражение `NOT (FALSE OR FALSE OR UNKNOWN)`, которое в свою очередь превращается в `NOT UNKNOWN` или просто `UNKNOWN`.

Результат `UNKNOWN` до применения к нему оператора `NOT` означает, что нельзя доподлинно установить, входит ли идентификатор клиента в множество, поскольку значение `NULL` может быть чем угодно, в том числе этим идентификатором. Вся хитрость заключается в том, что отрицание значения `UNKNOWN` тоже дает `UNKNOWN`, вследствие чего результат отбрасывается на этапе фильтрации. Если неизвестно, входит ли идентификатор в множество, мы не можем утверждать, что он туда не входит.

Проще говоря, если вы используете предикат `NOT IN` в сочетании с вложенным запросом, который возвращает хотя бы одно значение `NULL`, внешний код всегда будет возвращать пустой набор. Значения таблицы, которые точно встречаются в наборе, отсеиваются, поскольку внешний код должен вернуть записи, которых в этом наборе нет. Не попадают в результат и те строки, нахождение которых в наборе не определено, потому что нельзя сказать наверняка, входит ли значение в множество, одним из элементов которого является отметка `NULL`.

Как же избежать подобной проблемы?

Во-первых, если столбец не должен принимать значения `NULL`, нужно обязательно указать для него ограничение `NOT NULL`. Обеспечение целостности данных играет намного более важную роль, чем принято считать.

Во-вторых, при написании любых запросов необходимо помнить, что вы имеете дело с тремя значениями истинности (`TRUE`, `FALSE` и `UNKNOWN`). Обращайте внимание на возможность появления отметок `NULL` и на то, подходит ли ваш код для работы с ними (если нет, придется вносить изменения). В примере, который мы рассматривали выше, внешний запрос возвращает пустой набор, потому что на определенном этапе происходит сравнение со значением `NULL`. Если вы хотите узнать, входит ли идентификатор клиента в набор известных значений, игнорируя при этом отметки `NULL`, вам нужно исключить фактор неопределенности. Это можно сделать, добавив во вложенный запрос предикат `O.custid IS NOT NULL`, как показано ниже.

```
SELECT custid, companyname
FROM Sales.Customers
WHERE custid NOT IN(SELECT O.custid
                    FROM Sales.Orders AS O
                    WHERE O.custid IS NOT NULL);
```

Так вы явно исключите отметки `NULL`. Есть и другой способ: вместо предиката `NOT IN` можно использовать `NOT EXISTS`.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE NOT EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
```

Напомню, что в отличие от инструкции `IN` предикат `EXISTS` использует исчисление в двоичной системе. Он может вернуть либо `TRUE`, либо `FALSE` (зна-

чение UNKNOWN исключено). Когда вложенный запрос доходит до отметки NULL в атрибуте O.custid, выражение возвращает UNKNOWN и строка отсеивается. Предикат EXISTS попросту игнорирует значения NULL, как будто их не существует, обрабатывая только строки с определенными идентификаторами. Вместо NOT IN безопаснее использовать NOT EXISTS.

Закончив со всеми экспериментами, удалите лишнюю запись с помощью следующего кода.

```
DELETE FROM Sales.Orders WHERE custid IS NULL;
```

### Подстановка имен столбцов

Логические ошибки могут быть довольно неочевидными. В этом разделе я опишу проблемы, которые возникают при обычной подстановке имен столбцов во вложенном запросе. После того как вы поймете всю сложность ситуации, я покажу, как с ней справиться.

Запросы, представленные в этом разделе, обращаются к таблице MyShippers в схеме Sales. Чтобы создать такую таблицу и заполнить ее данными, запустите приведенный ниже код.

```
IF OBJECT_ID('Sales.MyShippers', 'U') IS NOT NULL
    DROP TABLE Sales.MyShippers;

CREATE TABLE Sales.MyShippers
(
    shipper_id INT NOT NULL,
    companyname NVARCHAR(40) NOT NULL,
    phone NVARCHAR(24) NOT NULL,
    CONSTRAINT PK_MyShippers PRIMARY KEY(shipper_id)
);

INSERT INTO Sales.MyShippers(shipper_id, companyname, phone)
VALUES(1, N'Поставщик GVSUA', N'(503) 555-0137'),
      (2, N'Поставщик ETYNR', N'(425) 555-0136'),
      (3, N'Поставщик ZHISN', N'(415) 555-0138');
```

Рассмотрим запрос, представленный ниже; он должен возвращать поставщиков, которые отправляли заказы клиенту под номером 43.

```
SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN
    (SELECT shipper_id
     FROM Sales.Orders
     WHERE custid = 43);
```

Вот какой результат мы получим.

shipper_id	companyname
1	Поставщик GVSUA
2	Поставщик ETYNR
3	Поставщик ZHISN

Очевидно, что только второй и третий поставщики отправляли товар клиенту с идентификатором 43, но по какой-то причине этот запрос вернул все содержимое таблицы **MyShippers**. Посмотрите внимательно на код и на схемы, которые в нем используются. Попробуйте найти логическое объяснение.

Оказывается, столбец, который хранит идентификаторы в таблице **Orders**, называется не **shipper\_id**, а **shipperid** (без символа подчеркивания). Столбец **shipper\_id** на самом деле находится в таблице **MyShippers**. Поиск по имени начинается в контексте вложенного запроса и только потом переходит на уровень выше. В нашем примере SQL Server ищет атрибут **shipper\_id** в таблице **Orders** и, не найдя его там, обращается к внешнему коду. В итоге атрибут берется из таблицы **MyShippers**.

Как видите, вместо автономного вложенного запроса случайно получился коррелирующий. И если в таблице **Orders** есть хоть одна строка, сравнение идентификаторов поставщиков всегда будет давать положительный результат, потому что вложенный запрос возвращает одни и те же идентификаторы для любого заказа.

Возможно, вам кажется, что это изъян языка SQL. Но члены комитета по разработке стандарта ANSI SQL вовсе не пытались завуалировать никаких «ошибок»; такая логика была предусмотрена специально, чтобы вы могли ссылаться на столбцы из внешних таблиц без всяких префиксов. И это действительно работает, если имена нужных вам столбцов встречаются только в одной таблице.

Такая проблема характерна для окружений, в которых при именовании атрибутов таблиц не соблюдается логика, например имена отличаются совсем незначительно: в одной таблице атрибут называется **shipperid**, а в другой **shipper\_id** (как в вышеприведенном примере). Этого достаточно, чтобы ошибка дала о себе знать.

Чтобы не попадать в такие ситуации, нужно соблюдать два простых правила, одно из которых ориентировано на долгосрочную перспективу, а второе можно начать использовать прямо сейчас.

Во-первых, сотрудники вашей организации должны понимать, насколько важной является последовательность при выборе имен для столбцов разных таблиц. Конечно, вряд ли вы сможете быстро поменять уже имеющиеся названия, не нарушая работоспособности программного кода.

Во-вторых, при использовании вложенных запросов в качестве префиксов для столбцов можно указывать псевдоним исходной таблицы. Благодаря этому поиск столбца всегда будет выполняться в заданном пространстве имен, и если его там не окажется, запрос вернет ошибку. Попробуйте, к примеру, запустить следующий код.

```
SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN
  (SELECT O.shipper_id
   FROM Sales.Orders AS O
   WHERE O.custid = 43);
```

Ниже показана ошибка, которую вы должны получить.

```
Сообщение 207, уровень 16, состояние 1, строка 4
Недопустимое имя столбца "shipper_id".
```

Увидев такое сообщение, вы, конечно же, сможете определить причину проблемы и исправить свой код.

```
SELECT shipper_id, companyname
FROM Sales.MyShippers
WHERE shipper_id IN
    (SELECT O.shipperid
     FROM Sales.Orders AS O
     WHERE O.custid = 43);
```

На этот раз запрос возвращает то, чего мы ожидали.

```
shipper_id  companyname
-----
2           Поставщик ETYNR
3           Поставщик ZHISN
```

Закончив с примерами, очистите БД с помощью приведенного ниже кода.

```
IF OBJECT_ID('Sales.MyShippers', 'U') IS NOT NULL
    DROP TABLE Sales.MyShippers;
```

## В заключение

Эта глава была посвящена вложенным запросам — как автономным (независимым от внешнего кода), так и коррелирующим (которые не способны работать самостоятельно). Вы узнали, что вложенные запросы могут возвращать скалярные и множественные значения. Здесь также содержится дополнительный раздел, в котором мы обсудили возвращение предыдущих и следующих значений, использование текущих агрегатов и решение проблем, которые возникают при работе с вложенными запросами. Всегда учитывайте особенности троичной логики и не забывайте указывать псевдонимы исходных таблиц в качестве префиксов для имен столбцов.

В следующей главе мы поговорим о вложенных табличных запросах, которые еще называются табличными выражениями.

## Упражнения

Здесь собраны упражнения, которые помогут вам лучше понять материал, рассмотренный в этой главе. Все запросы выполняются к объектам БД TSQL2012.

### 1

Напишите запрос, который возвращает все заказы, сделанные в последний день, учтенный в таблице `Orders`.

- Используется таблица `Sales.Orders`.

- Ожидаемый результат:

orderid	orderdate	custid	empid
11077	2008-05-06 00:00:00.000	65	1
11076	2008-05-06 00:00:00.000	9	4
11075	2008-05-06 00:00:00.000	68	8
11074	2008-05-06 00:00:00.000	73	7

## 2 (углубленное, по желанию)

Напишите запрос, который будет возвращать все заказы, размещенные клиентом (или клиентами) с самым большим количеством заказов. Учитывайте вероятность того, что одно и то же количество заказов может быть сразу у нескольких клиентов.

- Используется таблица **Sales.Orders**.
- Ожидаемый результат:

custid	orderid	orderdate	empid
71	10324	2006-10-08 00:00:00.000	9
71	10393	2006-12-25 00:00:00.000	1
71	10398	2006-12-30 00:00:00.000	2
71	10440	2007-02-10 00:00:00.000	4
71	10452	2007-02-20 00:00:00.000	8
71	10510	2007-04-18 00:00:00.000	6
71	10555	2007-06-02 00:00:00.000	6
71	10603	2007-07-18 00:00:00.000	8
71	10607	2007-07-22 00:00:00.000	5
71	10612	2007-07-28 00:00:00.000	1
71	10627	2007-08-11 00:00:00.000	8
71	10657	2007-09-04 00:00:00.000	2
71	10678	2007-09-23 00:00:00.000	7
71	10700	2007-10-10 00:00:00.000	3
71	10711	2007-10-21 00:00:00.000	5
71	10713	2007-10-22 00:00:00.000	1
71	10714	2007-10-22 00:00:00.000	5
71	10722	2007-10-29 00:00:00.000	8
71	10748	2007-11-20 00:00:00.000	3
71	10757	2007-11-27 00:00:00.000	6
71	10815	2008-01-05 00:00:00.000	2
71	10847	2008-01-22 00:00:00.000	4
71	10882	2008-02-11 00:00:00.000	4
71	10894	2008-02-18 00:00:00.000	1
71	10941	2008-03-11 00:00:00.000	7
71	10983	2008-03-27 00:00:00.000	2
71	10984	2008-03-30 00:00:00.000	1
71	11002	2008-04-06 00:00:00.000	4
71	11030	2008-04-17 00:00:00.000	7
71	11031	2008-04-17 00:00:00.000	6
71	11064	2008-05-01 00:00:00.000	1

(строк обработано: 31)

3

Напишите запрос, который возвращает список сотрудников, не обрабатывавших заказы 1 мая 2008 г.

- Используются таблицы `HR.Employees` и `Sales.Orders`.
- Ожидаемый результат:

empid	FirstName	lastname
3	Джуди	Лью
5	Свен	Бак
6	Пол	Суурс
9	Зоя	Долгопятова

4

Получите список стран, в которых есть клиенты, но нет сотрудников.

- Используются таблицы `HR.Employees` и `Sales.Customers`.
- Ожидаемый результат:

country
Австрия
Аргентина
Бельгия
Бразилия
Венесуэла
Германия
Дания
Ирландия
Испания
Италия
Канада
Мексика
Норвегия
Польша
Португалия
Финляндия
Франция
Швейцария
Швеция

(строк обработано: 19)

5

Напишите запрос, который возвращает все заказы, размещенные в последний день активности каждого клиента.

- Используется таблица `Sales.Orders`.



- Ожидаемый результат (в сокращенном виде):

custid	orderid	orderdate	empid
1	11011	2008-04-09 00:00:00.000	3
2	10926	2008-03-04 00:00:00.000	4
3	10856	2008-01-28 00:00:00.000	3
4	11016	2008-04-10 00:00:00.000	9
5	10924	2008-03-04 00:00:00.000	3
...			
87	11025	2008-04-15 00:00:00.000	6
88	10935	2008-03-09 00:00:00.000	4
89	11066	2008-05-01 00:00:00.000	7
90	11005	2008-04-07 00:00:00.000	2
91	11044	2008-04-23 00:00:00.000	4

(строк обработано: 90)

## 6

Напишите запрос, который возвращает список клиентов, размещавших заказы в 2007-м, но не в 2008 г.

- Используются таблицы `Sales.Orders` и `Sales.Customers`.
- Ожидаемый результат:

custid	companyname
21	Клиент KIDPX
23	Клиент WVFAF
33	Клиент FVXPQ
36	Клиент LVJSO
43	Клиент UISOJ
51	Клиент PVDZC
85	Клиент ENQZT

(строк обработано: 7)

## 7 (углубленное, по желанию)

Напишите запрос, который возвращает список клиентов, заказывавших товар под номером 12.

- Используются таблицы `Sales.Orders`, `Sales.Customers` и `Sales.OrderDetails`.
- Ожидаемый результат:

custid	companyname
48	Клиент DVFMB
39	Клиент GLLAG
71	Клиент LCOUJ
65	Клиент NYUHS
44	Клиент OXFRU
51	Клиент PVDZC

```
86      Клиент SNXOJ
20      Клиент THHDP
90      Клиент XBBVR
46      Клиент XPNIK
31      Клиент YJCBX
87      Клиент ZHYOS
```

(строк обработано: 12)

## 8 (углубленное, по желанию)

Напишите запрос, который вычисляет общее текущее количество заказанного товара для каждого клиента за каждый месяц.

- Используется таблица `Sales.CustOrders`.
- Ожидаемый результат (в сокращенном виде):

custid	ordermonth	qty	runqty
1	2007-08-01 00:00:00.000	38	38
1	2007-10-01 00:00:00.000	41	79
1	2008-01-01 00:00:00.000	17	96
1	2008-03-01 00:00:00.000	18	114
1	2008-04-01 00:00:00.000	60	174
2	2006-09-01 00:00:00.000	6	6
2	2007-08-01 00:00:00.000	18	24
2	2007-11-01 00:00:00.000	10	34
2	2008-03-01 00:00:00.000	29	63
3	2006-11-01 00:00:00.000	24	24
3	2007-04-01 00:00:00.000	30	54
3	2007-05-01 00:00:00.000	80	134
3	2007-06-01 00:00:00.000	83	217
3	2007-09-01 00:00:00.000	102	319
3	2008-01-01 00:00:00.000	40	359
...			

(строк обработано: 636)

## Решения

Здесь приводятся решения для упражнений из данного раздела.

### 1

Сначала напишите автономный вложенный запрос, возвращающий последнюю дату заказа в таблице `Orders`. Затем сошлитесь на его результат во внешнем коде (в инструкции `WHERE`), чтобы вернуть заказы, размещенные в этот день. Ниже приводится решение.

```
USE TSQL2012;

SELECT orderid, orderdate, custid, empid
FROM Sales.Orders
WHERE orderdate =
      (SELECT MAX(O.orderdate) FROM Sales.Orders AS O);
```

## 2

Эту проблему лучше всего решать в несколько этапов. Для начала нужно написать запрос, который возвращает клиента (или клиентов), разместившего наибольшее количество заказов. Чтобы это сделать, можно сгруппировать заказы по клиентам, отсортировать по функции COUNT (\*) и применить выражение TOP (1) WITH TIES, которое вернет идентификаторы нужных нам клиентов. Если вы уже забыли, как использовать параметр TOP, можете посмотреть в главе 2. Вот запрос, который выполняет первую часть условия.

```
SELECT TOP (1) WITH TIES O.custid
FROM Sales.Orders AS O
GROUP BY O.custid
ORDER BY COUNT(*) DESC;
```

Этот код возвращает идентификатор с номером 71, принадлежащий клиенту, который разместил наибольшее количество заказов (31). В таблице **Orders** нашелся только один такой клиент, но параметр WITH TIES, который мы использовали, перечисляет все найденные записи, поэтому наш код удовлетворяет заданному условию.

Второй этап заключается в том, чтобы получить из таблицы **Orders** все заказы, в которых идентификатор клиента совпадает с полученным ранее значением.

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders
WHERE custid IN
    (SELECT TOP (1) WITH TIES O.custid
     FROM Sales.Orders AS O
     GROUP BY O.custid
     ORDER BY COUNT(*) DESC);
```

## 3

Сначала нужно написать автономный вложенный запрос к таблице **Orders**, который отбирает заказы, размещенные начиная с 1 мая 2008 г., и возвращает идентификаторы сотрудников, которые значатся в этих заказах. Затем во внешнем коде надо выполнить запрос к таблице **Employees**, чтобы получить список сотрудников, идентификаторы которых не входят в набор, возвращенный вложенным запросом. Ниже представлено полноценное решение.

```
SELECT empid, FirstName, lastname
FROM HR.Employees
WHERE empid NOT IN
    (SELECT O.empid
     FROM Sales.Orders AS O
     WHERE O.orderdate >= '20080501');
```

## 4

Для начала необходимо написать автономный вложенный запрос к таблице **Employees**, который возвращает для каждой записи значение атрибута country. Затем во внешнем коде нужно обратиться к таблице **Customers**, отбирая только те

строки, атрибут `country` которых не входит в набор, полученный на предыдущем этапе. Укажите во внешней инструкции `SELECT` выражение `DISTINCT country`, чтобы убрать дубликаты, поскольку в одной и той же стране может находиться несколько клиентов. Полноценное решение показано ниже.

```
SELECT DISTINCT country
FROM Sales.Customers
WHERE country NOT IN
    (SELECT E.country FROM HR.Employees AS E);
```

## 5

Это упражнение похоже на первое, только здесь нас интересуют не просто заказы, сделанные в последний учетный день, но и клиенты, которые их разместили. Решения тоже имеют довольно много общего, но в данном случае нам придется сопоставлять идентификаторы клиентов, взятые из внешнего и вложенного кода, как показано ниже.

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders AS O1
WHERE orderdate =
    (SELECT MAX(O2.orderdate)
     FROM Sales.Orders AS O2
     WHERE O2.custid = O1.custid)
ORDER BY custid;
```

Дата заказа во внешней строке сравнивается не с последним днем, учетным в таблице `Orders`, а с последним днем активности текущего клиента.

## 6

Для решения этой задачи нам понадобятся вложенные запросы, а также предикаты `EXISTS` и `NOT EXISTS`; с их помощью мы получим список клиентов, размещавших заказы в 2007 г., и отбросим тех из них, у кого были заказы в 2008-м. Предикат `EXISTS` возвращает `TRUE` только при совпадении идентификаторов клиентов во внешней строке и таблице `Orders`; при этом учитываются только записи за 2007 г. Предикат `NOT EXISTS` возвращает `TRUE` при условии, что в таблице `Orders` нет записи с тем же идентификатором клиента, что и во внешней строке; заказы берутся за 2008 г. Ниже представлено полноценное решение.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS
    (SELECT *
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
          AND O.orderdate >= '20070101'
          AND O.orderdate < '20080101')
AND NOT EXISTS
    (SELECT *
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
          AND O.orderdate >= '20080101'
          AND O.orderdate < '20090101');
```

## 7

Чтобы решить это упражнение, мы воспользуемся предикатами EXISTS в сочетании с коррелирующими вложенными запросами. Внешний код должен обращаться к таблице **Customers**. Во внешней инструкции WHERE можно указать предикат EXISTS, внутри которого будет находиться коррелирующий вложенный запрос к таблице **Orders**; он будет возвращать заказы, относящиеся к текущему клиенту. Там же можно указать второй предикат EXISTS с еще одним вложенным запросом, но уже обращенным к таблице **OrderDetails**; он будет возвращать информацию о продукте с идентификатором 12. Таким образом, мы получим список клиентов, в заказах которых значится товар под номером 12. Ниже вы можете видеть полноценное решение.

```
SELECT custid, companyname
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT *
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
   AND EXISTS
     (SELECT *
      FROM Sales.OrderDetails AS OD
      WHERE OD.orderid = O.orderid
      AND OD.ProductID = 12));
```

## 8

Когда у меня возникают проблемы с написанием запроса, я обычно стараюсь перефразировать исходное условие так, чтобы его было удобнее выразить на языке T-SQL. Для начала можно немного изменить формулировку «общее текущее количество заказанного товара для каждого клиента за каждый месяц», сделать ее более формальной. Итак, мы должны вернуть для каждого клиента его идентификатор, месяц, общее количество заказанных товаров за этот месяц и сумму всех товаров, которые были заказаны до конца текущего месяца. Такое условие легко переводится на язык запросов.

```
SELECT custid, ordermonth, qty,
  (SELECT SUM(O2.qty)
   FROM Sales.CustOrders AS O2
   WHERE O2.custid = O1.custid
   AND O2.ordermonth <= O1.ordermonth) AS runqty
FROM Sales.CustOrders AS O1
ORDER BY custid, ordermonth;
```

## Глава 5

# ТАБЛИЧНЫЕ ВЫРАЖЕНИЯ

**Табличными** называют выражения, результат выполнения которых является корректным с точки зрения реляционной модели. Такие выражения можно подставлять в различные инструкции запроса вместо имен таблиц. Microsoft SQL Server поддерживает четыре разновидности табличных выражений, каждую из которых я подробно опишу в этой главе: производные таблицы, обобщенные табличные выражения (ОТВ), представления и встроенные функции с табличными значениями (ФТЗ). Все примеры будут касаться инструкции `SELECT`; использование табличных выражений в сочетании с другими командами мы рассмотрим в главе 8.

Результаты выполнения табличного выражения являются сугубо виртуальными — они нигде не сохраняются. При обращении к табличному выражению выполняется вложенный запрос. Другими словами, внешний и внутренний запросы объединяются в один, что позволяет напрямую работать с исходными объектами. Выгода от использования табличных выражений обычно касается логических аспектов кода, а не производительности. Они позволяют упростить решение задачи, используя модульный подход. Табличные выражения также помогают обойти некоторые ограничения языка, например, они позволяют ссылаться на псевдонимы столбцов, объявленные в инструкции `SELECT`, на этапах, предшествующих выполнению этой инструкции.

В конце данной главы мы рассмотрим оператор `APPLY`, который позволяет применять табличные выражения к каждой отдельной строке заданной таблицы.

## Производные таблицы

Производные таблицы (их еще называют **вложенными табличными запросами**) объявляются в инструкции `FROM` в рамках внешнего запроса, где они и существуют. Как только запрос завершает свою работу, производные таблицы исчезают.

Определение производной таблицы содержится внутри круглых скобок, а ее имя указывается с помощью инструкции `AS`. Например, следующий код объявляет производную таблицу `USACusts`, основанную на запросе, который возвращает всех американских клиентов; все ее содержимое извлекается с помощью внешней инструкции `SELECT`.

```
USE TSQL2012;

SELECT *
FROM (SELECT custid, companyname
      FROM Sales.Customers
      WHERE country = N'США') AS USACusts;
```

В этом простом примере мы могли бы обойтись и без производной таблицы, поскольку внешний запрос никак не обрабатывает полученные данные.

Вот какой результат получается.

custid	companyname
-----	-----
32	Клиент YSIQX
36	Клиент LVJSO
43	Клиент UISOJ
45	Клиент QXPPT
48	Клиент DVFMB
55	Клиент KZQZT
65	Клиент NYUHS
71	Клиент LCOUJ
75	Клиент XOJYP
77	Клиент LCYBZ
78	Клиент NLTPP
82	Клиент EYHKM
89	Клиент YBQTI

Чтобы запрос можно было считать корректным табличным выражением, он должен отвечать трем основным требованиям:

- Порядок не гарантируется. Табличное выражение должно возвращать реляционный результат, что подразумевает неупорядоченность строк. Напомню, что это одна из фундаментальных особенностей реляционной модели, относящаяся к теории множеств. В связи с этим стандартная разновидность языка SQL не поддерживает использование инструкции `ORDER BY` в запросах, с помощью которых объявляются табличные выражения; исключение составляют ситуации, когда данная инструкция не участвует в выводе конечного результата (например, в случае использования фильтра `OFFSET-FETCH`). Похожие правила действуют и в языке T-SQL, только к списку исключений добавляется еще и нестандартный фильтр `TOP`. Инструкция `ORDER BY` может служить частью определения выражений `TOP` или `OFFSET-FETCH`; в этом случае она занимается исключительно фильтрацией и не участвует в выводе конечного результата. Если инструкцию `ORDER BY` не указать в конце внешнего запроса, упорядоченность результирующего набора не гарантируется. Позже, в разделе «Представления и инструкция `ORDER BY`», мы еще вернемся к этой теме.
- У всех столбцов должны быть имена. Каждый столбец должен как-то называться; то есть любым табличным выражениям, которые объявляются в списке `SELECT`, нужно присваивать псевдонимы.
- Имена всех столбцов должны быть уникальными. Все столбцы в таблице следует называть по-разному. Табличное выражение, содержащее столбцы с одинаковыми именами, является некорректным. Такая ситуация возможна в том случае, когда в основе выражения лежит запрос, соединяющий две таблицы. Чтобы решить данную проблему, столбцам с одинаковыми именами можно назначить разные псевдонимы.

Эти требования вытекают из того факта, что табличное выражение должно возвращать реляционный результат. Всем атрибутам отношения следует дать имена;

имена атрибутов должны быть уникальными; тело отношения является множеством кортежей, а множество не может быть упорядоченным.

## Назначение псевдонимов для столбцов

Одно из преимуществ от использования табличных выражений заключается в том, что псевдонимы столбцов, объявленные во внутреннем запросе в инструкции `SELECT`, доступны и во внешнем коде. Это позволяет обойти ограничение, из-за которого псевдонимы, внесенные в список `SELECT`, нельзя использовать на предыдущих логических этапах (например, в инструкциях `WHERE` или `GROUP BY`).

Представьте, что вам нужно написать запрос к таблице `Sales.Orders`, который возвращает количество уникальных клиентов, обслуживаемых ежегодно. Пример решения, представленный ниже, является некорректным, поскольку инструкция `GROUP BY` ссылается на псевдоним, присвоенный в списке `SELECT`; при этом инструкция `GROUP BY` теоретически должна обрабатываться раньше.

```
SELECT
    YEAR(orderdate) AS orderyear,
    COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY orderyear;
```

При попытке запустить этот запрос вы получите ошибку.

Сообщение 207, уровень 16, состояние 1, строка 5  
Недопустимое имя столбца "orderyear".

Чтобы решить данную проблему, выражение `YEAR(orderdate)` можно указать сразу в двух инструкциях, `GROUP BY` и `SELECT`. Но что, если вам нужно работать с куда более длинным выражением? Скопировав его в разные места запроса, вы сделаете свой код менее понятным, усложните его сопровождение и получите потенциальный источник ошибок. Чтобы избежать всего этого, можно воспользоваться табличным выражением, как показано в листинге 5.1.

### Листинг 5.1. Запрос к производной таблице с использованием псевдонимов

```
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders) AS D
GROUP BY orderyear;
```

Результат будет следующим.

orderyear	numcusts
2006	67
2007	86
2008	81

Этот код определяет производную таблицу под названием `D`, основанную на запросе к таблице `Orders`, который возвращает из каждой записи год и идентификатор



заказа. В списке `SELECT` выражению `YEAR(orderdate)` назначается псевдоним `orderyear`, на который можно сослаться во внешних инструкциях `GROUP BY` и `SELECT` (с точки зрения внешнего кода он является столбцом таблицы `D`).

Как уже упоминалось ранее, `SQL Server` разворачивает табличное выражение и обращается к исходным объектам напрямую. В развернутом виде код из листинга 5.1 выглядит следующим образом.

```
SELECT YEAR(orderdate) AS orderyear, COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY YEAR(orderdate);
```

Это еще раз подчеркивает тот факт, что табличные выражения используются исключительно для логического структурирования кода. В целом они никак не влияют на производительность.

В листинге 5.1 для назначения псевдонима используется синтаксис вида <выражение> [AS] <псевдоним>. И хотя ключевое слово `AS` не является здесь обязательным, оно делает код более понятным, поэтому я советую вам его не игнорировать.

В некоторых случаях удобнее использовать альтернативный синтаксис. В нем имена столбцов перечисляются в скобках, сразу за названием табличного выражения (см. ниже).

```
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate), custid
      FROM Sales.Orders) AS D(orderyear, custid)
GROUP BY orderyear;
```

Рекомендуется использовать классический синтаксис, и тому есть несколько причин. При отладке, когда запрос, формирующий табличное выражение, выполняется отдельно, столбцы в результирующем наборе обозначаются с помощью выбранных вами псевдонимов. Альтернативный синтаксис не позволяет указать нужные вам имена столбцов, поэтому в результат попадут безымянные выражения. К тому же при использовании альтернативного синтаксиса будет сложнее понять, к какому выражению относится конкретный псевдоним, особенно если вы имеете дело с достаточно длинным табличным выражением.

Однако альтернативный синтаксис тоже может быть удобен. Например, если запрос, представляющий табличное выражение, не должен подвергаться дальнейшим изменениям и вы хотите работать с ним как с единым целым, чтобы во внешнем коде фигурировало только имя табличного выражения со списком столбцов. Говоря терминами традиционных языков программирования, альтернативный синтаксис позволяет описать интерфейс для взаимодействия между внешним запросом и табличным выражением.

## Использование аргументов

В запросе, который определяет производную таблицу, можно использовать аргументы, например локальные переменные или входящие параметры для таких операций, как функции или хранимые процедуры. Следующий код объявляет и инициализирует локальную переменную под названием `@empid`, ссылка на которую содержится в инструкции `WHERE` внутри запроса, представляющего производную таблицу `D`.

```

DECLARE @empid AS INT = 3;

SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM (SELECT YEAR(orderdate) AS orderyear, custid
      FROM Sales.Orders
      WHERE empid = @empid) AS D
GROUP BY orderyear;

```

Этот запрос возвращает годовое количество отдельных клиентов, заказы которых были обработаны заданным сотрудником (идентификатор последнего хранится в переменной @empid). Вот как выглядит результат.

orderyear	numcusts
2006	16
2007	46
2008	30

## Вложенность

Производные таблицы могут быть вложенными. Это происходит в ситуациях, когда определение данной таблицы размещено не отдельно, а внутри инструкции FROM внешнего запроса. Вложенность всегда являлась проблематичным аспектом программирования, из-за которого код становится более сложным и запутанным.

Рассмотрим запрос, представленный в листинге 5.2. Он возвращает годы, в которые делались заказы, и количество клиентов, обслуженных за каждый год (при условии, что это количество превышает 70 человек).

### Листинг 5.2. Запрос с вложенными производными таблицами

```

SELECT orderyear, numcusts
FROM (SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
      FROM (SELECT YEAR(orderdate) AS orderyear, custid
            FROM Sales.Orders) AS D1
      GROUP BY orderyear) AS D2
WHERE numcusts > 70;

```

Результат будет следующим.

orderyear	numcusts
2007	86
2008	81

Внутренняя производная таблица D1 нужна для того, чтобы присвоить выражению YEAR(orderdate) псевдоним orderyear. Запрос, направленный к D1, ссылается на столбец orderyear в инструкциях GROUP BY и SELECT, назначая псевдоним numcusts выражению COUNT(DISTINCT custid). Кроме того, он используется для определения производной таблицы D2. Запрос к D2 ссылается на столбец numcusts внутри инструкции WHERE, чтобы отобрать годы, в которые было обслужено больше 70 клиентов.

В этом примере производные таблицы упрощают решение, позволяя вместо дублирования целых выражений использовать одни и те же псевдонимы. Хотя,

учитывая ту сложность, которая появляется из-за вложенности, я не могу сказать с уверенностью, оптимален ли этот вариант. Ниже представлена реализация с повторным использованием выражений.

```
SELECT YEAR(orderdate) AS orderyear, COUNT(DISTINCT custid) AS numcusts
FROM Sales.Orders
GROUP BY YEAR(orderdate)
HAVING COUNT(DISTINCT custid) > 70;
```

В целом, вложенность является одним из наиболее проблемных аспектов производных таблиц.

## Множественные ссылки

С производными таблицами связана еще одна проблема: они определяются во внешней инструкции `FROM` и, как следствие, выполняются в контексте внешнего запроса, а не перед ним. В момент обработки внешней инструкции `FROM` производная таблица еще не существует, поэтому вы не можете ссылаться на нее больше одного раза — для этого вам придется разместить ее определение в разных участках одного и того же кода. Пример такой ситуации показан в листинге 5.3.

**Листинг 5.3.** Несколько производных таблиц, основанных на одном и том же запросе

```
SELECT Cur.orderyear,
       Cur.numcusts AS curnumcusts,
       Prv.numcusts AS prvnumcusts,
       Cur.numcusts - Prv.numcusts AS growth
FROM (SELECT YEAR(orderdate) AS orderyear,
             COUNT(DISTINCT custid) AS numcusts
      FROM Sales.Orders
      GROUP BY YEAR(orderdate)) AS Cur
LEFT OUTER JOIN
  (SELECT YEAR(orderdate) AS orderyear,
           COUNT(DISTINCT custid) AS numcusts
   FROM Sales.Orders
   GROUP BY YEAR(orderdate)) AS Prv
ON Cur.orderyear = Prv.orderyear + 1;
```

Запрос соединяет два экземпляра табличного выражения, создавая две производные таблицы `Cur` и `Prv`, которые представляют соответственно текущий и предыдущий годы. С помощью условия `Cur.orderyear = Prv.orderyear + 1` каждая строка из `Cur` сопоставляется с предыдущим годом из `Prev`. Поскольку это левое внешнее соединение, у первой записи не может быть предыдущего года. Внешняя инструкция `SELECT` вычисляет разницу между количеством клиентов, обслуженных за текущий и предыдущий годы.

Код, представленный в листинге 5.3, возвращает следующий результат.

orderyear	curnumcusts	prvnumcusts	growth
2006	67	NULL	NULL
2007	86	67	19
2008	81	86	-5

Из-за того, что вы не можете ссылаться на несколько экземпляров одной производной таблицы, приходится снова и снова описывать один и тот же запрос. В итоге код становится более длинным, сложным и предрасположенным к ошибкам.

## Обобщенные табличные выражения

ОТВ очень похожи на производные таблицы, но имеют несколько важных преимуществ.

Они определяются с помощью ключевого слова `WITH` и в целом выглядят следующим образом.

```
WITH <имя_выражения> [ (<список_задействованных_столбцов> ) ]
AS
(
    <внутренний_запро_определяющий_выражение>
)
<запрос_к_выражению>;
```

Внутренний запрос должен соответствовать всем требованиям, которые предъявляются к любому табличному выражению и которые мы уже перечислили ранее. Давайте рассмотрим простой пример. Следующий код определяет ОТВ под названием `USACusts`, основанное на запросе, который возвращает всех американских клиентов; все строки из `USACusts` отбираются внешним запросом.

```
WITH USACusts AS
(
    SELECT custid, companyname
    FROM Sales.Customers
    WHERE country = 'США'
)
SELECT * FROM USACusts;
```

Как и в случае с производными таблицами, результат ОТВ становится недоступным сразу после выполнения внешнего кода.

### ПРИМЕЧАНИЕ



В языке T-SQL ключевое слово `WITH` имеет несколько назначений. Когда с его помощью определяется ОТВ, предыдущая инструкция (если таковая имеется) завершается точкой с запятой — это позволяет избежать неоднозначности запроса. Но, как ни странно, в самом выражении точку с запятой использовать не обязательно, хотя я рекомендую указывать ее после всех выражений, независимо от контекста.

## Псевдонимы столбцов

ОТВ тоже поддерживают две формы синтаксиса для назначения псевдонимов. Первая выглядит как `<выражение> AS <псевдоним_столбца>`; второй список столбцов указывается в скобках сразу после названия выражения.

Ниже показан пример первой формы.

```
WITH C AS
(
    SELECT YEAR(orderdate) AS orderyear, custid
    FROM Sales.Orders
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

А вот как выглядит вторая форма.

```
WITH C(orderyear, custid) AS
(
    SELECT YEAR(orderdate), custid
    FROM Sales.Orders
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

То, в каких случаях лучше всего использовать каждый из этих синтаксисов, мы уже обсудили при рассмотрении производных таблиц.

## Использование аргументов

Как и в случае с производными таблицами, при определении ОТВ можно использовать аргументы. Ниже показан пример.

```
DECLARE @empid AS INT = 3;
WITH C AS
(
    SELECT YEAR(orderdate) AS orderyear, custid
    FROM Sales.Orders
    WHERE empid = @empid
)
SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
FROM C
GROUP BY orderyear;
```

## Определение нескольких ОТВ

На первый взгляд разница между производной таблицей и ОТВ является сугубо семантической. Но тот факт, что ОТВ используются уже после своего определения, дает им несколько важных преимуществ. Прежде всего, чтобы вызвать одно выражение из другого, необязательно делать их вложенными. Вместо этого в инструкции `WITH` следует объявить список разных ОТВ, разделяя их запятыми. Каждое ОТВ может ссылаться на предыдущее, и все они доступны во внешнем коде.

Например, следующий запрос является альтернативным вариантом листинга 5.2, в котором вместо вложенных производных таблиц используются ОТВ.

```

WITH C1 AS
(
    SELECT YEAR(orderdate) AS orderyear, custid
    FROM Sales.Orders
),
C2 AS
(
    SELECT orderyear, COUNT(DISTINCT custid) AS numcusts
    FROM C1
    GROUP BY orderyear
)
SELECT orderyear, numcusts
FROM C2
WHERE numcusts > 70;

```

Поскольку вы определили ОТВ перед его непосредственным использованием, вам не пришлось прибегать к вложенности. Каждое ОТВ размещается в запросе в виде отдельного модуля, что позволяет сделать код более легким для восприятия и сопровождения.

Формально вы не можете сделать ОТВ вложенными или объявить их внутри скобок, которые считаются частью определения производной таблицы. Но, как вы уже знаете, вложенность влечет множество проблем, поэтому можете думать об этом не как о препятствии, а как о средстве, которое позволяет сделать код более прозрачным.

## Множественные ссылки

Тот факт, что ОТВ используются, будучи уже определенными, несет в себе еще одно преимущество. С точки зрения внешней инструкции FOR ОТВ уже существует, поэтому вы можете ссылаться на разные экземпляры одного и того же выражения. Например, следующий запрос является полным аналогом кода, представленного в листинге 5.3, только вместо производных таблиц в нем используются ОТВ.

```

WITH YearlyCount AS
(
    SELECT YEAR(orderdate) AS orderyear,
           COUNT(DISTINCT custid) AS numcusts
    FROM Sales.Orders
    GROUP BY YEAR(orderdate)
)
SELECT Cur.orderyear,
       Cur.numcusts AS curnumcusts, Prv.numcusts AS prvnumcusts,
       Cur.numcusts - Prv.numcusts AS growth
FROM YearlyCount AS Cur
LEFT OUTER JOIN YearlyCount AS Prv
ON Cur.orderyear = Prv.orderyear + 1;

```

Как видите, выражение YearlyCount объявлено только один раз, но внешняя инструкция FROM ссылается на него дважды — с помощью псевдонимов Cur и Prv. В отличие от производных таблиц ОТВ может пребывать в единственном экземпляре. В результате запрос становится более чистым, простым для восприятия и, как следствие, устойчивым к ошибкам.

Если вас интересует производительность, вспомните, что табличные выражения обычно никак не влияют на скорость работы, поскольку они являются виртуальными сущностями. Обе ссылки на ОТВ в предыдущем примере, будут возвращены в полноценное выражение. Внутри этого запроса происходит соединение двух экземпляров таблицы **Orders**, каждый из которых предварительно сканируется и агрегируется, тот же физический процесс наблюдается при использовании производных таблиц. Если обработка каждой ссылки требует много вычислительных ресурсов, вы должны сохранять полученные результаты во временной таблице или табличной переменной. В этой книге акцент делается на написании кода, а не на вопросах производительности; возможность многократного использования одного и того же определения делает ОТВ значительно более привлекательными по сравнению с производными таблицами.

## Рекурсивные ОТВ

В этом разделе речь пойдет о более сложных вопросах, связанных с ОТВ, поэтому данный материал можно считать факультативным.

ОТВ имеют одну уникальную особенность: они поддерживают рекурсию. Рекурсивные ОТВ определяются несколькими запросами, которые называются закрепленными и рекурсивными элементами (должен присутствовать как минимум один элемент каждого типа). Общая форма рекурсивного ОТВ выглядит следующим образом.

```
WITH <имя_ОТВ> [ (<список_столбцов> ) ]
AS
(
  <закрепленный_элемент>
  UNION ALL
  <рекурсивный_элемент>
)
<внешний_запрос_к_ОТВ>;
```

**Закрепленный элемент** — это запрос, который возвращает корректный реляционный результат и вызывается всего один раз; здесь нет никаких отличий от нерекурсивного табличного выражения.

**Рекурсивный элемент** — это запрос, который содержит ссылку на имя ОТВ (результатирующий набор, полученный на предыдущей итерации). При первом вызове рекурсивного элемента предшествующий результат представлен значениями, которые вернул закрепленный элемент. При каждом следующем вызове имя ОТВ представляет результат предыдущего выполнения рекурсивного элемента. В этой замкнутой цепочке отсутствует явное условие преждевременного завершения. Выход из рекурсии происходит в тот момент, когда рекурсивный элемент возвращает пустой набор или достигает какого-то предела.

Оба запроса должны быть совместимы по количеству и типу возвращаемых столбцов.

Ссылка на имя ОТВ во внешнем запросе представляет объединенные результирующие наборы, возвращаемые вызовами закрепленного и рекурсивного элементов (при этом первый вызывается всего один раз).

Если это ваше первое знакомство с рекурсивными ОТВ, подобное объяснение может показаться вам не совсем понятным. Давайте обратимся к конкретному примеру. Следующий код демонстрирует использование рекурсивных ОТВ для получения информации о сотруднике (Дон Функ, идентификатор 2) и обо всех его подчиненных (прямых или опосредованных).

```
WITH EmpsCTE AS
(
    SELECT empid, mgrid, firstname, lastname
    FROM HR.Employees
    WHERE empid = 2

    UNION ALL

    SELECT C.empid, C.mgrid, C.firstname, C.lastname
    FROM EmpsCTE AS P
    JOIN HR.Employees AS C
    ON C.mgrid = P.empid
)
SELECT empid, mgrid, firstname, lastname
FROM EmpsCTE;
```

Закрепленный элемент обращается к таблице **HR.Employees** и просто возвращает запись под номером 2.

```
SELECT empid, mgrid, firstname, lastname
FROM HR.Employees
WHERE empid = 2
```

Рекурсивный элемент соединяет ОТВ (которое представляет предыдущий результирующий набор) и таблицу **Employees**, возвращая работников, которые находятся в прямом подчинении у сотрудников, полученных на предыдущей итерации.

```
SELECT C.empid, C.mgrid, C.firstname, C.lastname
FROM EmpsCTE AS P
JOIN HR.Employees AS C
ON C.mgrid = P.empid
```

Другими словами, вызов рекурсивного элемента происходит многократно, и каждый раз возвращается очередной уровень подчиненных. При первом вызове мы получаем прямых подчиненных Дона Функа (это сотрудники под номерами 3 и 5). Второй вызов возвращает прямых подчиненных сотрудников 3 и 5 (их идентификаторы 4, 6, 7, 8 и 9). При третьем заходе подчиненных больше не остается; возвращается пустой набор, в результате чего рекурсия останавливается.

Ссылка на имя ОТВ, размещенная во внешнем запросе, представляет объединенные результирующие наборы, то есть сотрудника под номером 2 и всех его подчиненных.

Ниже представлен результат.

empid	mgrid	firstname	lastname
2	1	Дон	Функ
3	2	Джуди	Лью
5	2	Свен	Бак
6	5	Пол	Суурс



7	5	Рассел	Кинг
9	5	Зоя	Долгопятова
4	3	Иаиль	Пелед
8	3	Мария	Камерон

В случае логической ошибки в условии соединения или при возникновении проблем с возвращаемым результатом рекурсивный элемент может вызываться бесконечно. В качестве меры безопасности SQL Server по умолчанию позволяет выполнять не более 100 рекурсивных итераций. Код, превысивший этот лимит, завершается ошибкой. Вы можете изменить это ограничение, указав выражение `OPTION (MAXRECURSION n)` в конце внешнего запроса; `n` обозначает максимальное количество рекурсивных вызовов и находится в диапазоне от 0 до 32 767.

Если вы хотите полностью убрать ограничение, укажите `MAXRECURSION 0`. Стоит отметить, что SQL Server хранит промежуточные результаты, возвращаемые закрепленными и рекурсивными элементами, в рабочей таблице внутри БД `tempdb`; если вы уберете ограничение и ваш запрос выйдет из-под контроля, рабочая таблица начнет очень быстро увеличиваться в размере. Если БД `tempdb` больше не сможет расти (например, из-за нехватки свободного места на диске), ваш запрос завершится ошибкой.

## Представления

Те два вида табличных выражений, которые мы успели рассмотреть (производные таблицы и ОТВ), могут работать в рамках одного-единственного запроса. Они исчезают в момент, когда внешний код завершает работу. Это означает, что их нельзя использовать больше одного раза.

**Представления и встроенные ФТЗ** — это два вида табличных выражений, которые пригодны для многократного использования; их определения представляют собой объекты БД. После своего создания они помещаются на постоянное хранение в БД и могут быть удалены оттуда только отдельной командой.

В остальном представления и встроенные ФТЗ ведут себя так же, как производные таблицы и ОТВ. Например, выполняя запрос к представлению или ФТЗ, SQL Server разворачивает определение табличного выражения, обращаясь к исходным объектам напрямую, как мы видели раньше.

Сначала рассмотрим представления, а к ФТЗ перейдем в следующем разделе.

Как я уже упоминал, **представление** — это табличное выражение, пригодное для многократного использования и хранящееся в БД. Например, следующий код создает представление под названием `USACusts` в схеме `Sales` в рамках БД `TSQL2012`; оно будет возвращать всех клиентов, проживающих в США.

```
IF OBJECT_ID('Sales.USACusts') IS NOT NULL
    DROP VIEW Sales.USACusts;
GO
CREATE VIEW Sales.USACusts
AS
SELECT
    custid, companyname, contactname,
    contacttitle, address, city, region,
```

```
postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'США';
GO
```

Представления, так же как и производные таблицы и ОТВ, поддерживают две формы назначения псевдонимов для столбцов; первую вы можете видеть в предыдущем примере, а вторая подразумевает перечисление псевдонимов в скобках сразу после имени представления.

Итак, создав объект в БД, вы можете обращаться к нему, как к обычной таблице.

```
SELECT custid, companyname
FROM Sales.USACusts;
```

Вы можете управлять доступом к представлению (как и к любому другому объекту, к которому можно выполнить запрос) посредством полномочий; это касается прав на использование команд SELECT, INSERT, UPDATE и DELETE. Например, вы можете закрыть прямой доступ к исходным таблицам, разрешив обращаться к ним только через представления.

Стоит отметить, что использование выражения SELECT (\*) в контексте представлений имеет особое значение. После компиляции в представление нельзя будет добавить новые столбцы. Допустим, вы определили представление на основе запроса SELECT \* FROM dbo.T1 и на момент компиляции таблица T1 содержит два атрибута — col1 и col2. SQL Server сохраняет информацию об этих атрибутах в виде метаданных. Если добавить в таблицу новый столбец, он не будет доступен в представлении. Вы можете обновить метаданные посредством хранимых процедур sp\_refreshview или sp\_refreshsqlmodule, но чтобы избежать путаницы, при определении представления лучше перечислить все нужные вам столбцы. Если возникла необходимость пересмотреть определение представления при изменении структуры исходной таблицы, используйте команду ALTER VIEW.

## Инструкция ORDER BY

Запрос, определяющий представление, должен соответствовать всем требованиям, которые упоминались ранее в отношении производных таблиц. Представление не должно быть упорядочено, все его столбцы должны иметь уникальные имена. В этом разделе я раскрою тему упорядоченности, которая является принципиальным и крайне важным для понимания моментом.

Как вы помните, инструкцию ORDER BY, которая формирует конечный результат, нельзя использовать в запросах, определяющих табличные выражения, потому что в реляционном наборе строки не должны быть упорядоченными. Идея создания отсортированного представления является абсурдной, так как она нарушает основные свойства отношений, устанавливаемых реляционной моделью. Если вы хотите упорядочить результат, полученный из представления, укажите инструкцию ORDER BY во внешнем запросе, как показано ниже.

```
SELECT custid, companyname, region
FROM Sales.USACusts
ORDER BY region;
```

Попробуйте запустить код, приведенный ниже; он создает представление с инструкцией `ORDER BY`, которая сортирует результат.

```
ALTER VIEW Sales.USACusts
AS

SELECT
    custid, companyname, contactname,
    contacttitle, address, city, region,
    postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'США'
ORDER BY region;
GO
```

Вы получите следующую ошибку.

Сообщение 1033, уровень 15, состояние 1, процедура USACusts, строка 10  
Предложение `ORDER BY` не допускается в представлениях, встроенных функциях, производных таблицах, вложенных запросах и обобщенных табличных выражениях, если вместе с ним не указано предложение `TOP`, `OFFSET` или `FOR XML`.

Данное сообщение говорит о том, что SQL Server допускает использование инструкции `ORDER BY` лишь в связке с параметрами `TOP`, `OFFSET-FETCH` и `FOR XML`, где ее роль не ограничивается обычной сортировкой выводимых строк. Подобное правило действует даже в стандартной версии SQL; исключение делается только для параметра `OFFSET-FETCH`.

В связи с тем что инструкцию `ORDER BY` можно использовать в сочетании с параметрами `TOP/OFFSET-FETCH`, некоторые люди думают, что у них есть возможность создавать упорядоченные представления. Например, в следующем запросе используется выражение `TOP (100) PERCENT`.

```
ALTER VIEW Sales.USACusts
AS

SELECT TOP (100) PERCENT
    custid, companyname, contactname,
    contacttitle, address, city, region,
    postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'США'
ORDER BY region;
GO
```

Формально этот код является корректным и даже создает представление, но вы должны понимать, что в табличном выражении инструкция `ORDER BY` может применяться исключительно в качестве логического фильтра для параметра `TOP`. Если инструкцию `ORDER BY` не указать во внешнем запросе к представлению, SQL Server не сможет гарантировать упорядоченность результата.

К примеру, запустите следующий запрос.

```
SELECT custid, companyname, region
FROM Sales.USACusts;
```

Как видно из результата, строки не отсортированы по региону.

custid	companyname	region
32	Клиент YSIQX	OR
36	Клиент LVJSO	OR
43	Клиент UISOJ	WA
45	Клиент QXPPT	CA
48	Клиент DVFMB	OR
55	Клиент KZQZT	AK
65	Клиент NYUHS	NM
71	Клиент LCOUJ	ID
75	Клиент XOJYP	WY
77	Клиент LCYBZ	OR
78	Клиент NLTPP	MT
82	Клиент EYNKM	WA
89	Клиент YBQTI	WA

Иногда бывает так, что инструкция `ORDER BY` содержится только в определении представления, а во внешнем запросе — нет. В подобных случаях нельзя сказать наперед, будет ли конечный результат упорядоченным. Сортировка может быть выполнена с целью оптимизации, особенно если не используется выражение `TOP (100) PERCENT`. Идея, которую я хочу донести, заключается в том, что любой порядок следования строк считается корректным и нельзя ожидать, что результат будет отсортирован каким-то определенным образом. Чтобы упорядочить результирующий набор, инструкцию `ORDER BY` нужно указывать во внешнем запросе.

В SQL Server 2012 появился еще один вариант создания «отсортированных представлений», который заключается в использовании инструкции `OFFSET` в сочетании с параметром `0 ROWS` (и без инструкции `FETCH`).

```
ALTER VIEW Sales.USACusts
AS

SELECT
    custid, companyname, contactname,
    contacttitle, address, city, region,
    postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'США'
ORDER BY region
OFFSET 0 ROWS;
GO
```

Теперь, даже если я обращусь к этому табличному выражению и не укажу инструкцию `ORDER BY`, результат все равно будет упорядочен по региону. Но еще раз хочу подчеркнуть: не полагайтесь на сортировку в представлениях. Этот пример выдает ожидаемый результат лишь благодаря оптимизации. Если вам нужно гаранти-

ровать упорядоченность результирующего набора, извлекаемого из представления, используйте во внешнем запросе инструкцию `ORDER BY`.

Не путайте определение табличного выражения с обычным запросом. Даже использование инструкции `ORDER BY` в сочетании с параметрами `TOP` или `OFFSET-FETCH` не гарантирует упорядоченности результата в контексте представлений. В обычном же запросе инструкция `ORDER BY` может служить как в качестве фильтра для параметров `TOP` или `OFFSET-FETCH`, так и для вывода результирующего набора.

## Параметры

При создании или изменении представлений можно указывать разные параметры. Для этого используется инструкция `WITH`. Параметры `ENCRYPTION` и `SCHEMABINDING` задаются в заголовке представления, а `CHECK OPTION` — в его конце. Назначение каждого из них описывается в следующих разделах.

## ENCRYPTION

Параметр `ENCRYPTION` доступен при создании или изменении представлений, хранимых процедур, триггеров и пользовательских функций. Он сигнализирует о том, что текст определения объекта будет храниться в зашифрованном виде. Зашифрованный текст недоступен обычным пользователям и представлениям каталога; для его просмотра необходимо иметь специальные полномочия.

Прежде чем переходить к рассмотрению параметра `ENCRYPTION`, запустите следующий код, чтобы откатить представление `USACusts` к его первоначальному виду.

```
ALTER VIEW Sales.USACusts
AS

SELECT
    custid, companyname, contactname,
    contacttitle, address, city, region,
    postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'США';
GO
```

Чтобы получить определение представления, запустите функцию `OBJECT_DEFINITION`, как показано ниже.

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.USACusts'));
```

Поскольку мы пока не использовали параметр `ENCRYPTION`, текст определения доступен для просмотра. Вы получите следующий результат.

```
CREATE VIEW Sales.USACusts
AS

SELECT
    custid, companyname, contactname,
    contacttitle, address, city, region,
    postalcode, country, phone, fax
FROM Sales.Customers
```

```
WHERE country = N'США';
```

Теперь давайте изменим определение, добавив в него параметр `ENCRYPTION`.

```
ALTER VIEW Sales.USACusts WITH ENCRYPTION
AS
```

```
SELECT
    custid, companyname, contactname,
    contacttitle, address, city, region,
    postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'США';
GO
```

Попробуйте снова запустить функцию `OBJECT_DEFINITION`.

```
SELECT OBJECT_DEFINITION(OBJECT_ID('Sales.USACusts'));
```

На этот раз вы получите `NULL`.

Вместо функции `OBJECT_DEFINITION` для вывода текста определения объекта можно использовать хранимую процедуру `sp_helptext`. В случае с представлением `USACusts` это выглядит так.

```
EXEC sp_helptext 'Sales.USACusts';
```

Поскольку наше представление создано с использованием параметра `ENCRYPTION`, вместо текста определения вы получите следующее сообщение.

Текст для объекта "Sales.USACusts" зашифрован.

## SCHEMABINDING

Параметр `SCHEMABINDING` привязывает представления и пользовательские функции к структуре базовой таблицы. Он сигнализирует о том, что привязанные объекты и их атрибуты не могут быть сброшены или изменены.

Давайте добавим параметр `SCHEMABINDING` в представление `USACusts`.

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS
```

```
SELECT
    custid, companyname, contactname,
    contacttitle, address, city, region,
    postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'США';
GO
```

Теперь попытаемся сбросить столбец `Address` в таблице `Customers`.

```
ALTER TABLE Sales.Customers DROP COLUMN address;
```

Мы получим следующую ошибку.

Сообщение 5074, уровень 16, состояние 1, строка 1  
объект "USACusts" зависит от столбца "address".  
Сообщение 4922, уровень 16, состояние 9, строка 1  
Ошибка ALTER TABLE DROP COLUMN address, так как один или несколько  
объектов обращаются к данному столбцу.

Если бы не параметр SCHEMABINDING, мы могли выполнить подобные структурные изменения, равно как и удалить таблицу **Customers**. Это было бы чревато ошибками, поскольку при запросе к представлению у нас появилась бы возможность ссылаться на несуществующие объекты или столбцы. Параметр SCHEMABINDING помогает избежать подобных проблем.

Определение объекта, в котором используется параметр SCHEMABINDING, должно отвечать определенным требованиям. В инструкции SELECT нельзя использовать символ звездочку (\*); имена должны быть указаны для каждого столбца отдельно. Кроме того, названия объектов обязаны содержать схему. На самом деле это довольно универсальные правила, которыми можно руководствоваться и в других ситуациях.

К тому же, как нетрудно догадаться, создание объектов с использованием параметра SCHEMABINDING считается хорошим тоном.

## CHECK OPTION

Параметр CHECK OPTION предотвращает изменения, конфликтующие с фильтром представления (если таковой содержится в определении).

Запрос, определяющий представление **USACusts**, отбирает клиентов, у которых атрибут country равен N'США'. Наше определение пока еще не содержит параметра CHECK OPTION. Это означает, что вы можете добавлять через данное представление строки с каким-то другим значением атрибута country; можно также изменять существующие записи, указывая страну, отличную от США. Например, следующий код успешно вставляет в представление запись о клиенте из Великобритании, чья компания называется «Клиент ABCDE».

```
INSERT INTO Sales.USACusts(  
    companyname, contactname, contacttitle, address,  
    city, region, postalcode, country, phone, fax)  
VALUES(  
    N'Клиент ABCDE', N'Контакт ABCDE',  
    N'Должность ABCDE', N'Адрес ABCDE',  
    N'Лондон', NULL, N'12345', N'Великобритания',  
    N'012-3456789', N'012-3456789');
```

Строка была добавлена в таблицу **Customers** через представление. Но теперь, попытавшись найти ее в этом же представлении, вы получите пустой набор, поскольку фильтр отбирает только американских клиентов.

```
SELECT custid, companyname, country  
FROM Sales.USACusts  
WHERE companyname = N'Клиент ABCDE';
```

При поиске добавленной строки можно обратиться непосредственно к таблице **Customers**.

```
SELECT custid, companyname, country
```

```
FROM Sales.Customers
WHERE companyname = N'Клиент ABCDE';
```

Так вы сможете получить информацию о клиенте, потому что новая строка действительно была сохранена в БД.

custid	companyname	country
92	Клиент ABCDE	Великобритания

Аналогичным образом в таблицу будут записаны сведения об изменении страны клиента из США на какую-то другую. При этом новая информация не пройдет фильтр и, как следствие, не попадет в представление.

Чтобы предотвратить изменения, которые конфликтуют с фильтром представления, укажите в конце определения WITH CHECK OPTION.

```
ALTER VIEW Sales.USACusts WITH SCHEMABINDING
AS
```

```
SELECT
    custid, companyname, contactname,
    contacttitle, address, city, region,
    postalcode, country, phone, fax
FROM Sales.Customers
WHERE country = N'США'
WITH CHECK OPTION;
GO
```

Теперь попробуйте добавить строку, которая не может пройти фильтр.

```
INSERT INTO Sales.USACusts(
    companyname, contactname, contacttitle, address,
    city, region, postalcode, country, phone, fax)
VALUES(
    N'Клиент FGHIJ', N'Контакт FGHIJ',
    N'Должность FGHIJ', N'Адрес FGHIJ',
    N'Лондон', NULL, N'12345', N'Великобритания',
    N'012-3456789', N'012-3456789');
```

Вы получите следующую ошибку.

Сообщение 550, уровень 16, состояние 1, строка 1  
 Ошибка при попытке вставки или обновления, поскольку целевое представление либо указывает WITH CHECK OPTION, либо охватывает представление, которое указывает WITH CHECK OPTION, а одна или несколько строк, получающиеся при операции, не определены в рамках ограничения CHECK OPTION.  
 Выполнение данной инструкции было прервано.

В конце можете очистить БД, выполнив следующий код.

```
DELETE FROM Sales.Customers
WHERE custid > 91;

IF OBJECT_ID('Sales.USACusts') IS NOT NULL DROP VIEW Sales.USACusts;
```



## Встроенные функции с табличными значениями

**Встроенные ФТЗ** — это табличные выражения, которые поддерживают входящие параметры. Фактически, если отбросить формальности, их можно считать параметризованными представлениями.

Следующий код создает в БД TSQL2012 встроенную функцию с табличными значениями под названием **GetCustOrders**.

```
USE TSQL2012;
IF OBJECT_ID('dbo.GetCustOrders') IS NOT NULL
    DROP FUNCTION dbo.GetCustOrders;
GO
CREATE FUNCTION dbo.GetCustOrders
    (@cid AS INT) RETURNS TABLE
AS
RETURN
    SELECT orderid, custid, empid, orderdate,
           requireddate, shippeddate, shipperid,
           freight, shipname, shipaddress, shipcity,
           shipregion, shippostalcode, shipcountry
    FROM Sales.Orders
    WHERE custid = @cid;
GO
```

Эта встроенная ФТЗ возвращает все заказы, размещенные клиентом, чей идентификатор был указан посредством входящего параметра `@cid`. Обращение к данной функции выполняется так же, как и к обычным таблицам — с помощью элементов языка DML. Параметры указываются в скобках, которые следуют за названием функции. Кроме того, необходимо убедиться, что табличное выражение имеет псевдоним; это не всегда является обязательным требованием, но так вы сможете сделать свой код более понятным и избежать потенциальных ошибок.

Следующий запрос обращается к нашей функции, запрашивая у нее все заказы, которые были сделаны клиентом под номером 1.

```
SELECT orderid, custid
FROM dbo.GetCustOrders(1) AS O;
```

Результат представлен ниже.

orderid	custid
10643	1
10692	1
10702	1
10835	1
10952	1
11011	1

Встроенную ФТЗ, как и любую таблицу, можно использоваться как часть соединения. Например, следующий запрос будет соединять нашу функцию с таблицей **Sales.OrderDetails**, сопоставляя заказы клиента под номером 1 с их составляющими.

```
SELECT O.orderid, O.custid, OD.productid, OD.qty
FROM dbo.GetCustOrders(1) AS O
     JOIN Sales.OrderDetails AS OD
       ON O.orderid = OD.orderid;
```

Этот код вернет следующий результат.

orderid	custid	productid	qty
10643	1	28	15
10643	1	39	21
10643	1	46	2
10692	1	63	20
10702	1	3	6
10702	1	76	15
10835	1	59	15
10835	1	77	2
10952	1	6	16
10952	1	28	2
11011	1	58	40
11011	1	71	20

В конце можете очистить БД, выполнив такой запрос.

```
IF OBJECT_ID('dbo.GetCustOrders') IS NOT NULL
    DROP FUNCTION dbo.GetCustOrders;
```

## Оператор APPLY

Табличный оператор APPLY отличается широкими возможностями. Как и другие табличные операторы, он используется внутри инструкции FROM. У него есть две разновидности: CROSS APPLY и OUTER APPLY. Первая реализует один логический этап обработки, а вторая — два.



### ПРИМЕЧАНИЕ

Оператор APPLY не входит в спецификацию языка SQL; у него есть стандартный аналог под названием LATERAL, но он не поддерживается в SQL Server.

Оператор APPLY принимает две таблицы (я будут называть их левой и правой), вторая может быть табличным выражением — обычно это производная таблица или встроенная ФТЗ. Оператор CROSS APPLY реализует один логический этап обработки; он применяет правое табличное выражение к каждой строке левой таблицы, возвращая объединенный результирующий набор.

На первый взгляд может показаться, что оператор CROSS APPLY имеет много общего с перекрестным соединением, и это в некотором роде действительно так. Например, следующие два запроса возвращают одинаковые результаты.

```
SELECT S.shipperid, E.empid
FROM Sales.Shippers AS S
     CROSS JOIN HR.Employees AS E;
```

```
SELECT S.shipperid, E.empid
FROM Sales.Shippers AS S
CROSS APPLY HR.Employees AS E;
```

В отличие от соединения правое табличное выражение в операторе CROSS APPLY может представлять разные наборы записей для каждой строки из левой таблицы. Для этого определение производной таблицы в правой части должно ссылаться на атрибуты, указанные слева. Того же эффекта легко добиться и с помощью встроенной ФТЗ, передавая атрибуты из левой таблицы в качестве входящих аргументов.

Например, следующий код использует оператор CROSS APPLY для получения трех последних заказов, размещенных каждым клиентом.

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
CROSS APPLY
  (SELECT TOP (3) orderid, empid, orderdate, requireddate
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
   ORDER BY orderdate DESC, orderid DESC) AS A;
```

Можете считать, что табличное выражение A — это коррелирующий вложенный запрос. С точки зрения логических этапов обработки правая сторона оператора APPLY (в нашем случае это производная таблица) применяется к каждой строке таблицы Customers. Обратите внимание на упоминание атрибута C.custid в фильтре с левой стороны. Производная таблица возвращает три последних заказа для текущей строки, взятой слева. Поскольку производная таблица применяется к каждой такой строке, оператор CROSS APPLY возвращает три последних заказа для всех клиентов.

Ниже в сокращенном виде показан конечный результат этого запроса.

custid	orderid	orderdate
1	11011	2008-04-09 00:00:00.000
1	10952	2008-03-16 00:00:00.000
1	10835	2008-01-15 00:00:00.000
2	10926	2008-03-04 00:00:00.000
2	10759	2007-11-28 00:00:00.000
2	10625	2007-08-08 00:00:00.000
3	10856	2008-01-28 00:00:00.000
3	10682	2007-09-25 00:00:00.000
3	10677	2007-09-22 00:00:00.000
...		

(строка обработано: 263)

Как вы помните, в SQL Server 2012 вместо инструкции TOP можно использовать стандартный параметр OFFSET-FETCH (см. ниже).

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
CROSS APPLY
  (SELECT orderid, empid, orderdate, requireddate
   FROM Sales.Orders AS O
   WHERE O.custid = C.custid
   ORDER BY orderdate DESC, orderid DESC
   OFFSET 0 ROWS FETCH NEXT 3 ROWS ONLY)
```

```
ORDER BY orderdate DESC, orderid DESC
OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY) AS A;
```

Если результатом правого табличного выражения является пустой набор, оператор `CROSS APPLY` не возвращает соответствующую левую строку. Например, клиенты под номерами 22 и 57 не размещали заказов. В обоих случаях производная таблица возвращает пустой набор; следовательно, эти клиенты не попадают в конечный результат. Если вы не хотите выбрасывать из левой таблицы строки, для которых правое табличное выражение возвращает пустой набор, используйте вместо `CROSS APPLY` оператор `OUTER APPLY`. Этот оператор имеет второй логический этап, на котором строки из левой таблицы, не имеющие соответствия с правой стороны, становятся внешними и добавляются в результирующий набор; при этом всем атрибутам, взятым из правого табличного выражения, присваивается значение `NULL`. Это в некотором смысле похоже на добавление внешних строк при левом внешнем соединении.

Следующий код возвращает три последних заказа для каждого клиента, включая в итоговый результат клиентов, которые никогда ничего не заказывали.

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
  OUTER APPLY
    (SELECT TOP (3) orderid, empid, orderdate, requireddate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
     ORDER BY orderdate DESC, orderid DESC) AS A;
```

На этот раз клиенты, не имеющие заказов (идентификаторы 22 и 57), тоже попадают в результат.

custid	orderid	orderdate
1	11011	2008-04-09 00:00:00.000
1	10952	2008-03-16 00:00:00.000
1	10835	2008-01-15 00:00:00.000
2	10926	2008-03-04 00:00:00.000
2	10759	2007-11-28 00:00:00.000
2	10625	2007-08-08 00:00:00.000
3	10856	2008-01-28 00:00:00.000
3	10682	2007-09-25 00:00:00.000
3	10677	2007-09-22 00:00:00.000
...		
22	NULL	NULL
...		
57	NULL	NULL
...		

(строка обработано: 265)

Данный запрос можно переписать, используя вместо `TOP` параметр `OFFSET-FETCH`.

```
SELECT C.custid, A.orderid, A.orderdate
FROM Sales.Customers AS C
  OUTER APPLY
    (SELECT orderid, empid, orderdate, requireddate
     FROM Sales.Orders AS O
     WHERE O.custid = C.custid
```

```
ORDER BY orderdate DESC, orderid DESC
OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY) AS A;
```

Встроенные ФТЗ инкапсулируют подробности своей работы, поэтому они могут показаться вам более удобными в сравнении с производными таблицами. ФТЗ упрощают чтение и сопровождение кода. К примеру, следующий запрос создает встроенную функцию под названием `TopOrders`, которая возвращает `@n` последних заказов, сделанных клиентом с идентификатором `@custid`.

```
IF OBJECT_ID('dbo.TopOrders') IS NOT NULL
    DROP FUNCTION dbo.TopOrders;
GO
CREATE FUNCTION dbo.TopOrders
    (@custid AS INT, @n AS INT)
    RETURNS TABLE
AS
RETURN
    SELECT TOP (@n) orderid, empid, orderdate, requireddate
    FROM Sales.Orders
    WHERE custid = @custid
    ORDER BY orderdate DESC, orderid DESC;
GO
```

Используя вместо `TOP` параметр `OFFSET-FETCH`, можно заменить запрос, находящийся внутри функции, следующим кодом.

```
SELECT orderid, empid, orderdate, requireddate
FROM Sales.Orders
WHERE custid = @custid
ORDER BY orderdate DESC, orderid DESC
OFFSET 0 ROWS FETCH FIRST @n ROWS ONLY;
```

Теперь вы можете отказаться от использования производной таблицы в пользу новой функции.

```
SELECT
    C.custid, C.companyname,
    A.orderid, A.empid, A.orderdate, A.requireddate
FROM Sales.Customers AS C
    CROSS APPLY dbo.TopOrders(C.custid, 3) AS A;
```

Такой код будет лучше поддаваться анализу и дальнейшим изменениям. В реальном процессе обработки ничего не поменялось, потому что, как я уже говорил, определения табличных выражений разворачиваются, и в итоге SQL Server работает напрямую с исходными объектами.

## В заключение

Табличные выражения несомненно помогут вам упростить код, сделать его более удобным в сопровождении и скрыть логику исходных запросов. Если вас не интересует возможность повторного использования кода, вам подойдут ОТВ, которые имеют несколько преимуществ перед производными таблицами.

Во-первых, благодаря отсутствию вложенности ОТВ обладают более модульной структурой и лучше поддаются дальнейшим изменениям. Во-вторых, вы можете использовать несколько экземпляров одного и того же ОТВ — с производными таблицами этого делать нельзя.

Если вам нужны табличные выражения, пригодные для многократного использования, вам подойдут представления и встроенные ФТЗ. Первые являются оптимальным вариантом для задач, в которых не нужны входящие параметры; во всех остальных случаях лучше использовать ФТЗ.

Если вы хотите применить табличное выражение к каждой строке исходной таблицы и получить в результате объединенный результирующий набор, используйте оператор `APPLY`.

## Упражнения

Приведенные ниже упражнения помогут вам лучше понять материал, рассмотренный в этой главе. Все запросы, которые здесь указываются, требуют подключения к БД `TSQL2012`.

### 1-1

Напишите запрос, который возвращает максимальное значение столбца `orderdate` для каждого сотрудника.

- Используется таблица `Sales.Orders`.
- Ожидаемый результат:

<code>empid</code>	<code>maxorderdate</code>
-----	-----
3	2008-04-30 00:00:00.000
6	2008-04-23 00:00:00.000
9	2008-04-29 00:00:00.000
7	2008-05-06 00:00:00.000
1	2008-05-06 00:00:00.000
4	2008-05-06 00:00:00.000
2	2008-05-05 00:00:00.000
5	2008-04-22 00:00:00.000
8	2008-05-06 00:00:00.000

(строк обработано: 9)

### 1-2

Выразите запрос, приведенный в предыдущем упражнении, в виде производной таблицы. Выполните соединение полученного результата с таблицей `Orders`, чтобы вернуть для каждого сотрудника заказ с последней датой.

- Используется таблица `Sales.Orders`.

- Ожидаемый результат:

empid	orderdate	orderid	custid
9	2008-04-29 00:00:00.000	11058	6
8	2008-05-06 00:00:00.000	11075	68
7	2008-05-06 00:00:00.000	11074	73
6	2008-04-23 00:00:00.000	11045	10
5	2008-04-22 00:00:00.000	11043	74
4	2008-05-06 00:00:00.000	11076	9
3	2008-04-30 00:00:00.000	11063	37
2	2008-05-05 00:00:00.000	11073	58
2	2008-05-05 00:00:00.000	11070	44
1	2008-05-06 00:00:00.000	11077	65

(строк обработано: 10)

## 2-1

Напишите запрос, который вычисляет номер строки для каждого заказа, выполняя предварительную сортировку по столбцам `orderdate` и `orderid`.

- Используется таблица `Sales.Orders`.
- Ожидаемый результат (в сокращенном виде):

orderid	orderdate	custid	empid	rownum
10248	2006-07-04 00:00:00.000	85	5	1
10249	2006-07-05 00:00:00.000	79	6	2
10250	2006-07-08 00:00:00.000	34	4	3
10251	2006-07-08 00:00:00.000	84	3	4
10252	2006-07-09 00:00:00.000	76	4	5
10253	2006-07-10 00:00:00.000	34	3	6
10254	2006-07-11 00:00:00.000	14	5	7
10255	2006-07-12 00:00:00.000	68	9	8
10256	2006-07-15 00:00:00.000	88	3	9
10257	2006-07-16 00:00:00.000	35	4	10

...

(строк обработано: 830)

## 2-2

Напишите запрос, который возвращает строки с 11 по 20 с учетом сортировки по столбцам `orderdate` и `orderid`. Прибегните к ОТВ, чтобы инкапсулировать код из предыдущего упражнения.

- Используется таблица `Sales.Orders`.
- Ожидаемый результат:

orderid	orderdate	custid	empid	rownum
10258	2006-07-17 00:00:00.000	20	1	11
10259	2006-07-18 00:00:00.000	13	4	12
10260	2006-07-19 00:00:00.000	56	4	13

10261	2006-07-19 00:00:00.000	61	4	14
10262	2006-07-22 00:00:00.000	65	8	15
10263	2006-07-23 00:00:00.000	20	9	16
10264	2006-07-24 00:00:00.000	24	6	17
10265	2006-07-25 00:00:00.000	7	2	18
10266	2006-07-26 00:00:00.000	87	3	19
10267	2006-07-29 00:00:00.000	25	4	20

(строк обработано: 10)

### 3 (углубленное, по желанию)

Напишите запрос, который возвращает управленческую цепочку, связанную с Зоей Долгопятовой (сотрудник под номером 9). Задействуйте рекурсивное ОТВ.

- Используется таблица `HR.Employees`.
- Ожидаемый результат:

empid	mgrid	firstname	lastname
-----	-----	-----	-----
9	5	Зоя	Долгопятова
5	2	Свен	Бак
2	1	Дон	Функ
1	NULL	Сара	Дэвис

(строк обработано: 4)

### 4-1

Напишите представление, которое возвращает общий объем заказанной продукции для каждого сотрудника, разбитый по годам.

- Используются таблицы `Sales.Orders` и `Sales.OrderDetails`.
- При выполнении запроса

```
SELECT * FROM Sales.VEmpOrders ORDER BY empid, orderyear;
```

результат должен быть следующим:

empid	orderyear	qty
-----	-----	-----
1	2006	1620
1	2007	3877
1	2008	2315
2	2006	1085
2	2007	2604
2	2008	2366
3	2006	940
3	2007	4436
3	2008	2476
4	2006	2212
4	2007	5273
4	2008	2313



5	2006	778
5	2007	1471
5	2008	787
6	2006	963
6	2007	1738
6	2008	826
7	2006	485
7	2007	2292
7	2008	1877
8	2006	923
8	2007	2843
8	2008	2147
9	2006	575
9	2007	955
9	2008	1140

(строк обработано: 27)

## 4-2 (углубленное, по желанию)

Напишите запрос к представлению `Sales.VEmpOrders`, который возвращает общее текущее количество заказанных товаров для сотрудников, с разбиением по годам.

- Используется представление `Sales.VEmpOrders`.
- Ожидаемый результат:

empid	orderyear	qty	runqty
-----	-----	-----	-----
1	2006	1620	1620
1	2007	3877	5497
1	2008	2315	7812
2	2006	1085	1085
2	2007	2604	3689
2	2008	2366	6055
3	2006	940	940
3	2007	4436	5376
3	2008	2476	7852
4	2006	2212	2212
4	2007	5273	7485
4	2008	2313	9798
5	2006	778	778
5	2007	1471	2249
5	2008	787	3036
6	2006	963	963
6	2007	1738	2701
6	2008	826	3527
7	2006	485	485
7	2007	2292	2777
7	2008	1877	4654
8	2006	923	923
8	2007	2843	3766
8	2008	2147	5913
9	2006	575	575
9	2007	955	1530
9	2008	1140	2670

(строк обработано: 27)

## 5-1

Создайте встроенную функцию, которая в качестве аргументов принимает идентификатор поставщика (@supid AS INT) и произвольное число товаров (@n AS INT). Функция должна возвращать @n самых дорогих товаров, предоставленных заданным поставщиком.

- Используется таблица **Production.Products**.
- При выполнении запроса:

```
SELECT * FROM Production.TopProducts(5, 2);
```

результат должен быть следующим:

productid	productname	unitprice
12	Продукт OSFNS	38,00
11	Продукт QMVUN	21,00

(строк обработано: 2)

## 5-2

С помощью оператора **CROSS APPLY** и функции, созданной вами в упражнении 4-1, получите для каждого поставщика список из двух самых дорогих товаров.

- Ожидаемый результат (в сокращенном виде):

supplierid	companyname	productid	productname	unitprice
8	Поставщик BWGYE	20	Продукт QHFFP	81,00
8	Поставщик BWGYE	68	Продукт TBTBL	12,50
20	Поставщик CIYNM	43	Продукт ZZZHR	46,00
20	Поставщик CIYNM	44	Продукт VJIEO	19,45
23	Поставщик ELCRN	49	Продукт FPYPN	20,00
23	Поставщик ELCRN	76	Продукт JYGFE	18,00
5	Поставщик EQPNC	12	Продукт OSFNS	38,00
5	Поставщик EQPNC	11	Продукт QMVUN	21,00

...

(строк обработано: 55)

- В конце можете очистить БД, используя следующие код.

```
IF OBJECT_ID('Sales.VEmpOrders') IS NOT NULL
    DROP VIEW Sales.VEmpOrders;
IF OBJECT_ID('Production.TopProducts') IS NOT NULL
    DROP FUNCTION Production.TopProducts;
```

## Решения

Здесь приводятся решения для упражнений, представленных в предыдущем разделе.

## 1-1

Этот запрос является всего лишь подготовкой к решению следующего упражнения. Он должен возвращать последнюю дату заказа для каждого сотрудника.

```
USE TSQL2012;

SELECT empid, MAX(orderdate) AS maxorderdate
FROM Sales.Orders
GROUP BY empid;
```

## 1-2

Здесь с помощью запроса, описанного в предыдущем упражнении, нужно определить производную таблицу и соединить ее с таблицей **Orders**. Это позволит получить для каждого сотрудника последний обработанный заказ.

```
SELECT O.empid, O.orderdate, O.orderid, O.custid
FROM Sales.Orders AS O
     JOIN (SELECT empid, MAX(orderdate) AS maxorderdate
          FROM Sales.Orders
          GROUP BY empid) AS D
      ON O.empid = D.empid
     AND O.orderdate = D.maxorderdate;
```

## 2-1

Это подготовительный этап для выполнения следующего упражнения. Вы должны обратиться к таблице **Orders** и пронумеровать строки с учетом сортировки по столбцам **orderdate** и **orderid**, как показано ниже.

```
SELECT orderid, orderdate, custid, empid,
       ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum
FROM Sales.Orders;
```

## 2-2

Для решения этой задачи вам необходимо создать ОТВ, основанное на запросе из предыдущего упражнения, и отобрать из него строки с номерами в диапазоне от 11 до 20.

```
WITH OrdersRN AS
(
    SELECT orderid, orderdate, custid, empid,
           ROW_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum
    FROM Sales.Orders
)
SELECT * FROM OrdersRN WHERE rownum BETWEEN 11 AND 20;
```

Вам, наверное, интересно, для чего здесь понадобилось табличное выражение. Оконные функции (например, **ROW\_NUMBER**) допускаются только в инструкциях **SELECT** и **ORDER BY** — их нельзя применять непосредственно на этапе **WHERE**. Но с помощью табличного выражения вам не составит труда вызвать функцию **ROW\_NUMBER** в инструкции **SELECT** и сослаться на полученный столбец во внешней инструкции **WHERE**, используя псевдоним.

### 3

Вы уже знаете, как получать записи о сотруднике и его подчиненных на всех уровнях. Можете считать, что в этом упражнении вам нужно сделать то же самое, только наоборот. В качестве закрепленного элемента здесь выступает запрос, который возвращает запись о сотруднике под номером 9. Рекурсивный элемент соединяет ОТВ (назовем его C), представляющее дочернюю строку (подчиненного) на предыдущем уровне, с таблицей **Employees** (назовем ее P), которая представляет родительскую строку (начальника) уровнем выше. Таким образом, каждый вызов рекурсивного элемента возвращает запись о начальнике, пока цикл не дойдет до самой высокой должности (генеральный директор).

Ниже приводится полноценное решение.

```
WITH EmpsCTE AS
(
    SELECT empid, mgrid, firstname, lastname
    FROM HR.Employees
    WHERE empid = 9

    UNION ALL

    SELECT P.empid, P.mgrid, P.firstname, P.lastname
    FROM EmpsCTE AS C
    JOIN HR.Employees AS P
      ON C.mgrid = P.empid
)
SELECT empid, mgrid, firstname, lastname
FROM EmpsCTE;
```

### 4-1

Это подготовительный этап для выполнения следующего упражнения. Здесь вам необходимо определить представление на основе запроса, который соединяет таблицы **Orders** и **OrderDetails**, группирует строки по идентификатору сотрудника и году и возвращает общий объем заказанной продукции для каждой группы. Определение представления должно выглядеть следующим образом.

```
USE TSQL2012;
IF OBJECT_ID('Sales.VEmpOrders') IS NOT NULL
    DROP VIEW Sales.VEmpOrders;
GO
CREATE VIEW Sales.VEmpOrders
AS

SELECT
    empid,
    YEAR(orderdate) AS orderyear,
    SUM(qty) AS qty
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
      ON O.orderid = OD.orderid
GROUP BY
    empid,
    YEAR(orderdate);
GO
```

## 4-2

В данном упражнении вам нужно получить общее текущее количество заказанных товаров для каждого сотрудника, с разбиением по годам. Для этого вы можете выполнить запрос к представлению VEmpOrders (назовем его V1), который возвращает для каждой строки идентификатор сотрудника, год заказа и количество заказанного товара. В списке SELECT можно разместить вложенный запрос, обращенный ко второму экземпляру VEmpOrders (назовем его V2); он будет возвращать общее количество заказанного товара для всех строк, у которых идентификатор сотрудника совпадает со значением, полученным в V1, а год заказа меньше или равен дате из первого экземпляра VEmpOrders. Ниже представлено полноценное решение.

```
SELECT empid, orderyear, qty,
       (SELECT SUM(qty)
        FROM Sales.VEmpOrders AS V2
        WHERE V2.empid = V1.empid
          AND V2.orderyear <= V1.orderyear) AS runqty
FROM Sales.VEmpOrders AS V1
ORDER BY empid, orderyear;
```

В главе 7 вы познакомитесь с новыми методиками вычисления общего текущего количества с помощью оконных функций.

## 5-1

В этом упражнении необходимо определить функцию под названием TopProducts, которая должна возвращать @n самых дорогих товаров, предоставленных поставщиком с идентификатором @supid (@n и @supid являются входящими параметрами). Вот как может выглядеть это определение.

```
USE TSQL2012;
IF OBJECT_ID('Production.TopProducts') IS NOT NULL
    DROP FUNCTION Production.TopProducts;
GO
CREATE FUNCTION Production.TopProducts
    (@supid AS INT, @n AS INT)
    RETURNS TABLE
AS
RETURN
    SELECT TOP (@n) productid, productname, unitprice
    FROM Production.Products
    WHERE supplierid = @supid
    ORDER BY unitprice DESC;
GO
```

В SQL Server 2012 вместо TOP разрешено использовать параметр OFFSET-FETCH. Вы можете заменить внутренний запрос в своей функции следующим кодом.

```
SELECT productid, productname, unitprice
FROM Production.Products
WHERE supplierid = @supid
ORDER BY unitprice DESC
OFFSET 0 ROWS FETCH FIRST @n ROWS ONLY;
```

## 5-2

В этом упражнении вам нужно написать запрос к таблице `Production.Suppliers`, а затем с помощью оператора `CROSS APPLY` применить к каждой ее записи функцию, созданную на предыдущем этапе. В результате вы должны получить по два самых дорогих продукта у каждого поставщика. Решение приводится ниже.

```
SELECT S.supplierid, S.companyname, P.productid, P.productname,  
       P.unitprice  
FROM Production.Suppliers AS S  
     CROSS APPLY Production.TopProducts(S.supplierid, 2) AS P;
```

## Глава 6

# ОПЕРАТОРЫ РАБОТЫ С НАБОРАМИ

Данный вид операторов предназначен для работы с двумя входящими наборами (или мультимножествами, если использовать термины языка SQL), полученными в результате выполнения двух запросов. Как вы помните, мультимножество может содержать дубликаты, поэтому его нельзя считать настоящим множеством. Этот термин я буду использовать для описания промежуточных результатов работы входящих запросов. Операторы работы с наборами могут генерировать как настоящие множества (без дубликатов, как при использовании параметра `DISTINCT`), так и мультимножества (с дубликатами, как при использовании параметра `ALL`). В этой главе рассматриваются оба случая, хотя приоритет отдается работе с множествами.

Язык T-SQL поддерживает три оператора работы с наборами: `UNION`, `INTERSECT` и `EXCEPT`. Сначала мы познакомимся с общей формой и требованиями, которым они должны соответствовать, а затем рассмотрим каждый в отдельности.

Запрос, в котором используется оператор работы с наборами, в общем случае выглядит так.

```
Входящий запрос 1  
<оператор>  
Входящий запрос 2  
[ORDER BY ...]
```

Оператор построчно сравнивает результирующие наборы, которые были возвращены двумя входящими запросами. Попадание строки в конечный результат зависит как от самого сравнения, так и от разновидности оператора. Поскольку множества (или мультимножества) по определению не упорядочены, ни в одном из двух запросов не может быть инструкции `ORDER BY` (которая, как вы помните, сортирует результат, возвращая курсор). Тем не менее эту инструкцию можно применить к результату выполнения оператора постфактум.

Каждый отдельный запрос может включать любые логические этапы обработки, кроме инструкции `ORDER BY` (она используется только для вывода конечного результата). Ее можно задействовать в связке со значениями, которые получаются путем применения оператора к двум входящим результирующим наборам.

Количество столбцов, генерируемых двумя входящими запросами, должно совпадать. Их типы данных обязаны быть совместимыми, то есть значения автоматически приводятся к одному типу (с учетом приоритета).

Имена столбцов в итоговом результате берутся из первого запроса, в котором и следует присваивать все псевдонимы.

У оператора работы с наборами есть одна интересная особенность: при сравнении строк он исходит из того, что две отметки NULL являются тождественными. Позже вы увидите, почему это так важно.

В стандартной версии языка SQL у каждого оператора работы с наборами может присутствовать один из двух параметров — либо `DISTINCT` (по умолчанию), либо `ALL`. Первый убирает из итогового результата все дубликаты. В Microsoft SQL Server параметр `DISTINCT` поддерживается во всех трех операторах работы с наборами, а `ALL` только в `UNION`. При этом параметр `DISTINCT` нельзя указать явно; он применяется в том случае, когда отсутствует параметр `ALL`. Каждой разновидности операторов `UNION`, `INTERSECT` и `EXCEPT` будет посвящен отдельный подраздел.

## Объединение

**Объединением** двух множеств (назовем их *A* и *B*) является множество, содержащее все элементы из *A* и *B*. Другими словами, в результирующий набор попадают элементы, принадлежащие хотя бы одному из множеств. То, как происходит объединение, показано на рисунке 6.1 (такую схему еще называют **диаграммой Венна**). Закрашенная область представляет результат, возвращаемый оператором работы с наборами.

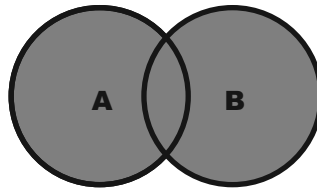


Рис. 6.1. Объединение двух множеств  $A \cup B$

В языке T-SQL оператор `UNION` объединяет результаты двух входящих запросов. Строка попадает в результирующий набор даже в том случае, если присутствует только с одной стороны. Этот оператор в T-SQL поддерживает оба параметра — `DISTINCT` (по умолчанию) и `ALL`.

### Мультимножества (оператор `UNION ALL`)

Оператор `UNION ALL` возвращает все строки, которые присутствуют в обоих входящих мультимножествах, не проводя при этом сравнения и не удаляя дубликаты. Допустим, Запрос1 возвращает *m* строк, а Запрос2 — *n* строк. Тогда результат работы выражения `Запрос1 UNION ALL Запрос2` будет содержать *m+n* строк.

В следующем коде оператор `UNION ALL` используется для объединения информации о местах проживания сотрудников и клиентов.

```
USE TSQ2012;  
  
SELECT country, region, city FROM HR.Employees  
UNION ALL  
SELECT country, region, city FROM Sales.Customers;
```



Результат, представленный ниже в сокращенном виде, будет содержать 100 строк — 9 из таблицы `Employees` и 91 из таблицы `Customers`.

country	region	city
-----	-----	-----
США	WA	Сиэтл
США	WA	Такома
США	WA	Киркланд
США	WA	Редмонд
Великобритания	NULL	Лондон
Великобритания	NULL	Лондон
Великобритания	NULL	Лондон
...		
Финляндия	NULL	Оулу
Бразилия	SP	Резенди
США	WA	Сиэтл
Финляндия	NULL	Хельсинки
Польша	NULL	Варшава

(строк обработано: 100)

Поскольку оператор `UNION ALL` не удаляет дубликаты, возвращаемый им результат является мультимножеством. Одна и та же строка может присутствовать в результирующем наборе в нескольких экземплярах (в нашем случае это Великобритания, NULL, Лондон).

## Множества (оператор `UNION`)

Оператор работы с множествами `UNION` (с параметром `DISTINCT` по умолчанию) объединяет результаты выполнения двух запросов, удаляя дубликаты. Если строка содержится в обоих наборах, в конечный результат она попадет только один раз; другими словами, итоговый набор является полноценным множеством.

К примеру, следующий код возвращает уникальные места проживания, которые относятся либо к сотрудникам, либо к клиентам.

```
SELECT country, region, city FROM HR.Employees
UNION
SELECT country, region, city FROM Sales.Customers;
```

В этом примере в отличие от предыдущего оператор `UNION` удаляет дубликаты. Следовательно, результат будет содержать 71 строку (как показано ниже).

country	region	city
-----	-----	-----
Австрия	NULL	Грац
Австрия	NULL	Зальцбург
Аргентина	NULL	Буэнос-Айрес
Бельгия	NULL	Брюссель
Бельгия	NULL	Шарлеруа
...		
Франция	NULL	Страсбург
Швейцария	NULL	Берн
Швейцария	NULL	Женева

Швеция	NULL	Брацке
Швеция	NULL	Лулео

(строк обработано: 71)

В каких ситуациях лучше всего использовать каждый из этих операторов? Если в результате объединения двух наборов могут возникать дубликаты, которые вы должны сохранить, используйте `UNION ALL`. Если нужны только уникальные строки, следует выбрать оператор `UNION`. Если дубликатов не может быть в принципе, вам подойдет любой вариант, хотя в этом случае я бы рекомендовал использовать оператор `UNION ALL`, так как он позволяет избежать накладных расходов, связанных с поиском одинаковых строк.

## Пересечение

**Пересечением** двух множеств (назовем их  $A$  и  $B$ ) является множество всех элементов, которые принадлежат как  $A$ , так и  $B$ . Эта операция схематически изображена на рисунке 6.2.

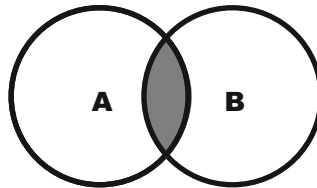


Рис. 6.2. Пересечение двух множеств  $A \cap B$

В языке T-SQL оператор `INTERSECT` возвращает пересечение результирующих наборов, полученных при выполнении двух запросов; в результат попадают только те строки, которые содержатся в обоих входящих наборах. Сначала мы рассмотрим оператор `INTERSECT` (с параметром `DISTINCT` по умолчанию), затем перейдем к его аналогу, `INTERSECT ALL`, который возвращает мультимножество (эта разновидность еще не реализована в SQL Server 2012).

## Множества (оператор `INTERSECT`)

Первым делом оператор `INTERSECT` убирает дубликаты из двух входящих мультимножеств (превращая их в настоящие множества) и затем возвращает строки, которые содержатся в обоих наборах. Таким образом, если строка попадает в результат, это означает, что она как минимум по одному разу входит в каждое мультимножество.

Например, следующий код возвращает уникальные места проживания, которые являются общими для сотрудников и клиентов.

```
SELECT country, region, city FROM HR.Employees
INTERSECT
SELECT country, region, city FROM Sales.Customers;
```

Результат представлен ниже.

country	region	city
-----	-----	-----
Великобритания	NULL	Лондон
США	WA	Киркланд
США	WA	Сиэтл

Не важно, сколько раз упоминается конкретное место проживания сотрудника или клиента. Чтобы попасть в результирующий набор, оно должно присутствовать в таблицах **Employees** и **Customers** хотя бы в одном экземпляре. Как видно по результату этого запроса, всего три места проживания относятся одновременно и к сотруднику, и к клиенту.

Ранее я уже упоминал, что при сравнении строк операторы работы с наборами считают любые две отметки NULL равными. Есть сотрудники и клиенты с местоположением Великобритания, NULL, Лондон, но причина, по которой они появляются в результирующем наборе, может показаться не совсем очевидной. Атрибуты **country** и **city** не допускают неопределенных значений, поэтому их сравнение происходит довольно прозрачно. А вот с атрибутом **region** все немного сложнее; когда оператор работы с наборами сравнивает две отметки NULL, относящиеся к сотруднику и клиенту, он считает их равными, поэтому строка, которая их содержит, попадает в конечный результат.

В случаях, когда такой подход к сравнению значений NULL вас устраивает (как в предыдущем примере), операторы работы с наборами не имеют достойных альтернатив. К примеру, вместо оператора **INTERSECT** можно использовать внутреннее соединение или предикат **EXISTS**. В обоих случаях при сравнении отметок NULL в атрибуте **region** вы бы получали значение UNKNOWN, что приведет к отбрасыванию строк с такими значениями. Без дополнительных манипуляций ни внутреннее соединение, ни предикат **EXISTS** не вернули бы строку Великобритания, NULL, Лондон, несмотря на то что она содержится с обеих сторон.

## Мультимножества (оператор **INTERSECT ALL**)

Этот дополнительный раздел ориентирован на читателей, у которых не возникло проблем с предыдущими темами, представленными в главе. Согласно спецификации языка SQL оператор **INTERSECT** должен поддерживать параметр **ALL**, но такая возможность пока не реализована в SQL Server 2012. Позже мы рассмотрим аналоги этой связки, доступные в языке T-SQL.

Как вы помните, связка с оператором **UNION** и параметром **ALL** возвращала все одинаковые строки. Почти то же самое происходит в случае с пересечением. Отличие лишь в том, что выражение **INTERSECT ALL** сокращает количество дубликатов в соответствии с тем мультимножеством, в котором их меньше всего. Другими словами, оператор **INTERSECT ALL** не только учитывает факт существования строки с обеих сторон, но и следит за количеством экземпляров этой строки в каждом входящем наборе. Все выглядит так, будто поиск соответствия выполняется по каждому экземпляру строки. Если в первом мультимножестве содержится  $x$  экземпляров строки **R**, а во втором —  $y$ , тогда в результирующий набор строка **R** попадет  $\text{minimum}(x, y)$  раз. Например, строка Великобритания, NULL, Лондон упоминается в таблицах **Employees** и **Customers** четыре и шесть раз соответственно; учитывая места проживания

сотрудников и клиентов, оператор `INTERSECT ALL` должен вернуть четыре экземпляра этой строки, поскольку с точки зрения логики мы имеем четыре пересечения.

Несмотря на то что в `SQL Server` нет встроенной поддержки оператора `INTERSECT ALL`, вы можете воспользоваться альтернативным решением, благодаря которому добьетесь того же результата. Речь идет о функции `ROW_NUMBER`, которая позволяет нумеровать экземпляры каждой строки во всех входящих запросах. Чтобы передать ей список атрибутов, необходимо использовать инструкцию `PARTITION BY`, а посредством выражения `(SELECT <константа>)` внутри инструкции `ORDER BY` можно сигнализировать о том, что порядок следования строк не имеет значения.



**ПРИМЕЧАНИЕ**

Выражение `ORDER BY (SELECT <константа>)` внутри оконной функции — это один из способов сообщить `SQL Server` о том, что результат не должен быть упорядочен. Ядро БД понимает, что вовсе не обязательно тратить лишние ресурсы на сортировку, поскольку всем строкам будет назначена одна и та же константа.

После этого оператор `INTERSECT` нужно применить к двум запросам с функцией `ROW_NUMBER`. Так как экземпляры всех строк пронумерованы, пересечение будет основываться не только на исходных атрибутах, но и на порядковых номерах. Например, каждый экземпляр строки Великобритания, `NULL`, Лондон в таблице `Employees` будет иметь номер от 1 до 4. В таблице `Customers` та же строка встречается шесть раз, поэтому для всех ее вхождений будут использоваться номера от 1 до 6. В итоге пересекутся четыре первых экземпляра.

Ниже представлено полноценное решение.

```
SELECT
    ROW_NUMBER()
        OVER(PARTITION BY country, region, city
             ORDER BY (SELECT 0)) AS rownum,
    country, region, city
FROM HR.Employees

INTERSECT

SELECT
    ROW_NUMBER()
        OVER(PARTITION BY country, region, city
             ORDER BY (SELECT 0)),
    country, region, city
FROM Sales.Customers;
```

Этот код генерирует следующий результат.

rownum	country	region	city
1	Великобритания	NULL	Лондон
1	США	WA	Киркланд
1	США	WA	Сиэтл
2	Великобритания	NULL	Лондон
3	Великобритания	NULL	Лондон
4	Великобритания	NULL	Лондон

Номера здесь являются вспомогательным инструментом, и оператор `INTERSECT ALL` не должен их возвращать. Чтобы оставить в результирующем наборе только исходные атрибуты, можно определить `OTB`, взяв за основу имеющийся запрос. Ниже показан пример, как посредством оператора `INTERSECT ALL` можно получить все места проживания, которые являются общими как для сотрудников, так и для клиентов.

```
WITH INTERSECT_ALL
AS
(
    SELECT
        ROW_NUMBER()
        OVER(PARTITION BY country, region, city
             ORDER BY (SELECT 0)) AS rownum,
        country, region, city
    FROM HR.Employees

    INTERSECT

    SELECT
        ROW_NUMBER()
        OVER(PARTITION BY country, region, city
             ORDER BY (SELECT 0)),
        country, region, city
    FROM Sales.Customers
)
SELECT country, region, city
FROM INTERSECT_ALL;
```

Результат выполнения этого запроса идентичен тому, который можно было бы получить, прибегнув стандартному оператору `INTERSECT ALL`.

country	region	city
Великобритания	NULL	Лондон
США	WA	Киркланд
США	WA	Сиэтл
Великобритания	NULL	Лондон
Великобритания	NULL	Лондон
Великобритания	NULL	Лондон

Разность

**Разностью** множеств  $A$  и  $B$  ( $A - B$ ) является множество элементов, принадлежащих  $A$ , но не принадлежащих  $B$ . Можете считать, что это все элементы  $A$ , кроме тех, которые также входят в  $B$ . На рисунке 6.3 схематически изображена разность  $A - B$ .

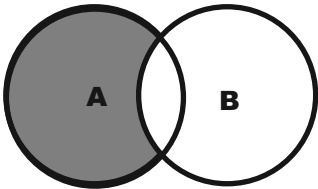


Рис. 6.3. Разность множеств  $A - B$

В языке T-SQL разность множеств реализована в виде оператора работы с наборами под названием EXCEPT; он принимает два входящих запроса и возвращает строки, которые присутствуют в первом из них, но отсутствуют во втором. Сначала мы рассмотрим оператор EXCEPT (с параметром DISTINCT по умолчанию), затем перейдем к его аналогу — INTERSECT ALL, поддержка которого еще не реализована в SQL Server 2012.

## Множества (оператор EXCEPT)

Первым делом оператор EXCEPT убирает дубликаты из двух входящих множеств (превращая их тем самым в настоящие множества), после чего возвращает строки, которые содержатся в первом наборе, но которых нет во втором. Другими словами, строка, попавшая в конечный результат, встречается в первом множестве как минимум в одном экземпляре, а во втором ее точно нет. Оператор EXCEPT, в отличие от предыдущих двух, является ассиметричным, то есть учитывает порядок следования входящих множеств.

Очередной код возвращает отдельные места проживания, которые относятся исключительно к сотрудникам (но не к клиентам).

```
SELECT country, region, city FROM HR.Employees
EXCEPT
SELECT country, region, city FROM Sales.Customers;
```

Результатом выполнения этого запроса будут две записи.

country	region	city
США	WA	Редмонд
США	WA	Такома

Код, представленный ниже, напротив, отбирает места проживания клиентов, которые не имеют отношения к сотрудникам.

```
SELECT country, region, city FROM Sales.Customers
EXCEPT
SELECT country, region, city FROM HR.Employees;
```

В результате мы получим 66 записей.

country	region	city
Австрия	NULL	Грац
Австрия	NULL	Зальцбург
Аргентина	NULL	Буэнос-Айрес
Бельгия	NULL	Брюссель
Бельгия	NULL	Шарлеруа
...		
Франция	NULL	Страсбург
Швейцария	NULL	Берн
Швейцария	NULL	Женева
Швеция	NULL	Брацке
Швеция	NULL	Лулео

(строка обработано: 66)

У оператора EXCEPT есть несколько альтернатив. Вместо него можно применить внешнее соединение, которое оставляет только внешние строки (те, что появляются лишь с определенной стороны). Еще один вариант — использование предиката NOT EXISTS, но если вам нужно, чтобы любые две отметки NULL считались тождественными по умолчанию, оператор EXCEPT будет оптимальным выбором.

## Мультимножества (оператор EXCEPT ALL)

Этот дополнительный раздел ориентирован на читателей, у которых не возникло проблем с предыдущими темами, представленными в этой главе. Оператор EXCEPT ALL делает почти то же самое, что и EXCEPT, но при этом учитывает количество экземпляров каждой строки. Если в первом мультимножестве содержится  $x$  экземпляров строки  $R$ , а во втором  $y$ , и  $x > y$ , тогда выражение `Query1 EXCEPT ALL Query2` вернет  $x - y$  строк  $R$ . Проще говоря, оператор EXCEPT ALL возвращает все копии одной и той же строки из первого набора, у которых нет соответствий во втором.

SQL Server не предоставляет встроенную поддержку оператора EXCEPT ALL, но, как и в случае с INTERSECT ALL, у вас есть возможность воспользоваться альтернативным вариантом. Пронумеруйте экземпляры каждой строки с помощью функции ROW\_NUMBER и затем примените оператор EXCEPT к двум входящим запросам. В результирующий набор попадут только те экземпляры, для которых не будет найдено соответствий.

В следующем примере показано, как с помощью оператора EXCEPT ALL получить все места проживания, которые касаются сотрудников, но не имеют отношения к клиентам.

```
WITH EXCEPT_ALL
AS
(
    SELECT
        ROW_NUMBER()
            OVER(PARTITION BY country, region, city
                ORDER BY (SELECT 0)) AS rownum,
        country, region, city
    FROM HR.Employees

    EXCEPT

    SELECT
        ROW_NUMBER()
            OVER(PARTITION BY country, region, city
                ORDER BY (SELECT 0)),
        country, region, city
    FROM Sales.Customers
)
SELECT country, region, city
FROM EXCEPT_ALL;
```

Результат выполнения этого запроса представлен ниже.

country	region	city
США	WA	Редмонд
США	WA	Такома
США	WA	Сиэтл

# Приоритет

В языке SQL операторы работы с наборами имеют определенный приоритет. Например, оператор `INTERSECT` всегда выполняется первым, а `UNION` и `EXCEPT` считаются равными и запускаются в порядке своего размещения внутри запроса.

Первоочередность оператора `INTERSECT` проиллюстрирована на примере следующего кода.

```
SELECT country, region, city FROM Production.Suppliers
EXCEPT
SELECT country, region, city FROM HR.Employees
INTERSECT
SELECT country, region, city FROM Sales.Customers;
```

По причинам, изложенным выше, оператор `INTERSECT` вызывается перед `EXCEPT`, хотя в коде запроса он указан вторым. Таким образом, этот запрос можно перефразировать так: «места проживания, которые относятся к поставщикам, кроме тех, что имеют отношение (и к сотрудникам, и к клиентам)».

Результат представлен ниже.

country	region	city
Австралия	NSW	Сидней
Австралия	Виктория	Мельбурн
Бразилия	NULL	Сан-Паулу
Великобритания	NULL	Манчестер
Германия	NULL	Берлин
Германия	NULL	Куксхафен
Германия	NULL	Франкфурт
Дания	NULL	Лингби
Испания	Астурия	Овьедо
Италия	NULL	Равенна
Италия	NULL	Салерно
Канада	Квебек	Монреаль
Канада	Квебек	Сент-Иасент
Нидерланды	NULL	Зандам
Норвегия	NULL	Сандвика
Сингапур	NULL	Сингапур
США	LA	Новый Орлеан
США	MA	Бостон
США	MI	Энн-Арбор
США	OR	Бенд
Финляндия	NULL	Лаппеэнранта
Франция	NULL	Анси
Франция	NULL	Монсо
Франция	NULL	Париж
Швеция	NULL	Гетеборг
Швеция	NULL	Стокгольм
Япония	NULL	Осака
Япония	NULL	Токио

(строк обработано: 28)



Для управления порядком вызова операторов работы с наборами можно использовать круглые скобки, так как они имеют самый высокий приоритет. К тому же скобки делают код более понятным и снижают вероятность возникновения ошибок. Например, если вы хотите получить «места проживания, которые относятся к поставщикам, но не к сотрудникам и которые также являются местами проживания клиентов», используйте следующий код.

```
(SELECT country, region, city FROM Production.Suppliers
EXCEPT
SELECT country, region, city FROM HR.Employees)
INTERSECT
SELECT country, region, city FROM Sales.Customers;
```

Результат выполнения этого запроса показан ниже.

country	region	city
-----	-----	-----
Германия	NULL	Берлин
Канада	Квебек	Монреаль
Франция	NULL	Париж

## Эмуляция неподдерживаемых логических этапов

Учитывая целевую аудиторию книги, данный материал можно считать углубленным и необязательным к прочтению. Отдельные запросы, которые передаются операторам работы с наборами, могут содержать любые логические этапы обработки (например, табличные операторы, инструкции WHERE, GROUP BY и HAVING), кроме ORDER BY. Но ORDER BY также является единственным этапом, который можно применять к результатам выполнения этих операторов. Что же делать, если вам необходимо выполнить какой-то другой этап? Этого нельзя сделать напрямую во внешнем запросе, который запускает оператор, однако подобное ограничение можно обойти. Внешний запрос необходимо оформить в виде табличного выражения и потом уже применять к нему любые логические этапы обработки. Например, следующий код возвращает количество отдельных мест проживания сотрудников или клиентов в каждой стране.

```
SELECT country, COUNT(*) AS numlocations
FROM (SELECT country, region, city FROM HR.Employees
UNION
SELECT country, region, city FROM Sales.Customers) AS U
GROUP BY country;
```

Результат выполнения этого запроса представлен ниже.

country	numlocations
-----	-----
Австрия	2
Аргентина	1
Бельгия	2
Бразилия	4
Великобритания	2
Венесуэла	4

Германия	11
Дания	2
Ирландия	1
Испания	3
Италия	3
Канада	3
Мексика	1
Норвегия	1
Польша	1
Португалия	1
США	14
Финляндия	2
Франция	9
Швейцария	2
Швеция	2

(строк обработано: 21)

На примере этого запроса проиллюстрировано применение логического этапа обработки GROUP BY к результату выполнения оператора UNION; аналогичным образом можно было бы применить любой другой этап.

Тот факт, что инструкцию ORDER BY нельзя использовать в отдельных запросах, к которым применяется оператор работы с наборами, также нередко является причиной определенных логических проблем. Представьте, что вам необходимо ограничить количество строк в данных запросах с помощью параметров TOP или OFFSET-FETCH. Безусловно, это реально сделать посредством табличного выражения, которое содержит параметры TOP или OFFSET-FETCH в своем определении. В таком случае инструкция ORDER BY будет всего лишь частью фильтра и не повлияет на вывод конечного результата.

Итак, чтобы передать оператору работы с наборами запрос, в котором используются параметры TOP или OFFSET-FETCH, нужно просто определить табличное выражение, что наряду с внешним запросом будет играть роль входящего параметра для этого оператора. Например, следующий код использует параметр TOP, чтобы получить два последних заказа для каждого сотрудника с идентификатором от 3 до 5.

```
SELECT empid,orderid,orderdate
FROM (SELECT TOP (2) empid,orderid,orderdate
      FROM Sales.Orders
      WHERE empid = 3
      ORDER BY orderdate DESC,orderid DESC) AS D1

UNION ALL

SELECT empid,orderid,orderdate
FROM (SELECT TOP (2) empid,orderid,orderdate
      FROM Sales.Orders
      WHERE empid = 5
      ORDER BY orderdate DESC,orderid DESC) AS D2;
```

Ниже показан результат.

empid	orderid	orderdate
-----	-----	-----
3	11063	2008-04-30 00:00:00.000

3	11057	2008-04-29 00:00:00.000
5	11043	2008-04-22 00:00:00.000
5	10954	2008-03-17 00:00:00.000

Этот запрос можно переписать с использованием параметра `OFFSET-FETCH`.

```
SELECT empid,orderid,orderdate
FROM (SELECT empid,orderid,orderdate
      FROM Sales.Orders
      WHERE empid = 3
      ORDER BY orderdate DESC,orderid DESC
      OFFSET 0 ROWS FETCH FIRST 2 ROWS ONLY) AS D1

UNION ALL

SELECT empid,orderid,orderdate
FROM (SELECT empid,orderid,orderdate
      FROM Sales.Orders
      WHERE empid = 5
      ORDER BY orderdate DESC,orderid DESC
      OFFSET 0 ROWS FETCH FIRST 2 ROWS ONLY) AS D2;
```

## В заключение

Эта глава посвящена операторам работы с наборами. Мы обсудили их базовый синтаксис и предъявляемые требования, а также рассмотрели каждый из них в отдельности — `UNION`, `INTERSECT` и `EXCEPT`. Вы узнали, что стандартная версия SQL поддерживает две разновидности каждого оператора — `DISTINCT` (множество) и `ALL` (мультимножество), но при этом в SQL Server 2012 параметр `ALL` реализован только для оператора `UNION`. Я также показал, как, используя функцию `ROW_NUMBER` и табличные выражения можно компенсировать отсутствие поддержки операторов `INTERSECT ALL` и `EXCEPT ALL`. В конце мы обсудили тему приоритетов и то, каким образом можно эмулировать не поддерживаемые логические этапы обработки, используя все те же табличные выражения.

## Упражнения

В этом разделе представлены упражнения, которые помогут вам закрепить материал, пройденный в главе 6. Все упражнения, кроме первого, требуют подключения к демонстрационной БД `TSQL2012`.

### 1

Напишите запрос, который генерирует виртуальную вспомогательную таблицу с десятью числами в диапазоне от 1 до 10, не используя при этом цикл. Конечный результат можно не упорядочивать.

- Не используется ни одной таблицы.

- Ожидаемый результат:

```
n
-----
1
2
3
4
5
6
7
8
9
10
```

(строк обработано: 10)

## 2

Напишите запрос, который возвращает пары «клиент — сотрудник», работавшие с заказами в январе, а не в феврале 2008 г.

- Используется таблица **Sales.Orders**.
- Ожидаемый результат:

custid	empid
-----	-----
1	1
3	3
5	8
5	9
6	9
7	6
9	1
12	2
16	7
17	1
20	7
24	8
25	1
26	3
32	4
38	9
39	3
40	2
41	2
42	2
44	8
47	3
47	4
47	8
49	7
55	2
55	3
56	6

59	8
63	8
64	9
65	3
65	8
66	5
67	5
70	3
71	2
75	1
76	2
76	5
80	1
81	1
81	3
81	4
82	6
84	1
84	3
84	4
88	7
89	4

(строк обработано: 50)

### 3

Напишите запрос, который возвращает пары «клиент — сотрудник», работавшие с заказами как в январе, так и в феврале 2008 г.

- Используется таблица **Sales.Orders**.
- Ожидаемый результат:

custid	empid
-----	-----
20	3
39	9
46	5
67	1
71	4

(строк обработано: 5)

### 4

Напишите запрос, который возвращает пары «клиент — сотрудник», работавшие с заказами в январе и феврале 2008 г., но не в 2007 г.

- Используется таблица **Sales.Orders**.
- Ожидаемый результат:

custid	empid
-----	-----
67	1
46	5

(строк обработано: 2)

## 5 (углубленное, по желанию)

У вас есть следующий запрос.

```
SELECT country, region, city
FROM HR.Employees

UNION ALL

SELECT country, region, city
FROM Production.Suppliers;
```

Вы должны добавить в него код, благодаря которому строки из таблицы **Employees** будут находиться в результирующем наборе перед строками из таблицы **Suppliers**. Кроме того, каждый сегмент должен быть отсортирован по стране, региону и городу.

- Используются таблицы **HR.Employees** и **Production.Suppliers**.
- Ожидаемый результат:

country	region	city
-----	-----	-----
Великобритания	NULL	Лондон
Великобритания	NULL	Лондон
Великобритания	NULL	Лондон
Великобритания	NULL	Лондон
США	WA	Киркланд
США	WA	Редмонд
США	WA	Сиэтл
США	WA	Сиэтл
США	WA	Такома
Австралия	NSW	Сидней
Австралия	Виктория	Мельбурн
Бразилия	NULL	Сан-Паулу
Великобритания	NULL	Лондон
Великобритания	NULL	Манчестер
Германия	NULL	Берлин
Германия	NULL	Куксхафен
Германия	NULL	Франкфурт
Дания	NULL	Лингби
Испания	Астурия	Овьедо
Италия	NULL	Равенна
Италия	NULL	Салерно
Канада	Квебек	Монреаль
Канада	Квебек	Сент-Иасент
Нидерланды	NULL	Зандам
Норвегия	NULL	Сандвика
Сингапур	NULL	Сингапур
США	LA	Новый Орлеан
США	MA	Бостон
США	MI	Энн-Арбор
США	OR	Бенд
Финляндия	NULL	Лаппеэнранта
Франция	NULL	Анси
Франция	NULL	Монсо
Франция	NULL	Париж
Швеция	NULL	Гетеборг
Швеция	NULL	Стокгольм

Япония	NULL	Осака
Япония	NULL	Токио

(строк обработано: 38)

## Решения

Здесь приводятся решения для упражнений, представленных в главе 6.

### 1

В языке T-SQL на этапе `SELECT` вместо инструкции `FROM` можно использовать константы. В таких случаях возвращается таблица с единственной строкой. Например, следующий код вернет запись с одним столбцом под названием `n`, в котором хранится значение 1.

```
SELECT 1 AS n;
```

Вот как будет выглядеть результат.

```
n
-----
1
```

(строк обработано: 1)

Вы можете сгенерировать несколько таких результирующих наборов с числами от 1 до 10 и объединить их помощью оператора `UNION ALL`, как показано ниже.

```
SELECT 1 AS n
UNION ALL SELECT 2
UNION ALL SELECT 3
UNION ALL SELECT 4
UNION ALL SELECT 5
UNION ALL SELECT 6
UNION ALL SELECT 7
UNION ALL SELECT 8
UNION ALL SELECT 9
UNION ALL SELECT 10;
```



### ПРИМЕЧАНИЕ

SQL Server поддерживает улучшенную инструкцию `VALUES`, принцип работы которой вы можете наблюдать на примере команды `INSERT`. Инструкция `VALUES` не ограничена выводом всего одной строки; она может возвращать любое количество записей. Кроме того, ее можно использовать не только в рамках команды `INSERT`, но и в табличных выражениях, строки которых основаны на константах. К примеру, вот как бы вы могли решить это упражнение, используя инструкцию `VALUES` вместо оператора `UNION ALL`.

```
SELECT n
FROM (VALUES(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)) AS Nums(n);
```

Подробно об инструкции `VALUES` и конструкторе табличных значений я расскажу при обсуждении команды `INSERT` в главе 8.

## 2

Вы можете решить это упражнение с помощью оператора `EXCEPT`. Запрос, находящийся слева, будет возвращать пары «клиент — сотрудник», которые работали с заказами в январе 2008 г. Пары, возвращаемые правым запросом, будут относиться к февралю 2008 г. Итоговое решение должно выглядеть так.

```
USE TSQL2012;

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20080101' AND orderdate < '20080201'

EXCEPT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20080201' AND orderdate < '20080301';
```

## 3

В этом упражнении в отличие от предыдущего нам нужно отбирать пары «клиент — сотрудник», которые занимались заказами в оба периода. Поэтому вместо оператора `EXCEPT` мы должны использовать `INTERSECT`.

```
SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20080101' AND orderdate < '20080201'

INTERSECT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20080201' AND orderdate < '20080301';
```

## 4

Здесь придется применить сразу несколько операторов работы с наборами. Как и в предыдущем упражнении, нам понадобится оператор `INTERSECT`, чтобы получить пары «клиент — сотрудник», которые работали с заказами в январе и феврале 2008 г. Мы также должны убрать из результирующего набора те пары, которые проявляли активность в 2007 г.; для этого между предыдущим результатом и третьим запросом следует вставить оператор `EXCEPT`. Итоговое решение будет выглядеть следующим образом.

```
SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20080101' AND orderdate < '20080201'

INTERSECT

SELECT custid, empid
FROM Sales.Orders
```



```
WHERE orderdate >= '20080201' AND orderdate < '20080301'

EXCEPT
SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate < '20080101';
```

Не забывайте, что оператор `INTERSECT` имеет приоритет перед оператором `EXCEPT`. В данном случае нам подходит стандартный порядок выполнения операторов, поэтому здесь нет нужды что-либо менять. С помощью круглых скобок вы сделаете код более «прозрачным» (см. ниже).

```
(SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20080101' AND orderdate < '20080201'

INTERSECT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20080201' AND orderdate < '20080301')

EXCEPT

SELECT custid, empid
FROM Sales.Orders
WHERE orderdate >= '20070101' AND orderdate < '20080101';
```

## 5

С этим упражнением связан один интересный момент: отдельные запросы не могут содержать инструкцию `ORDER BY`, и на то есть веская причина. Чтобы решить данную проблему, можно добавить результирующий набор каждого запроса в столбец (назовем его `sortcol`), содержимое которого будет основано на константе. Имейте в виду, что для запроса, направленного к таблице `Employees`, значение константы должно быть меньше, чем в случае с таблицей `Suppliers`. После этого на основе внешнего запроса, в котором вызывается оператор, необходимо создать табличное выражение и указать в его инструкции `ORDER BY` столбцы `sortcol`, `country`, `region` и `city` (именно в таком порядке). Полноценное решение представлено ниже.

```
SELECT country, region, city
FROM (SELECT 1 AS sortcol, country, region, city
      FROM HR.Employees

      UNION ALL

      SELECT 2, country, region, city
      FROM Production.Suppliers) AS D
ORDER BY sortcol, country, region, city;
```

## Глава 7

# ПРОДВИНУТЫЕ ЗАПРОСЫ

Начинаем с подробного рассмотрения оконных функций, которые позволяют в гибкой и эффективной манере применять результаты вычислений к наборам строк. Далее ознакомимся с методиками по разворачиванию данных. **Разворачивание** — это преобразование строк в столбцы. Существует также обратная операция, превращающая столбцы в строки. В конце мы затронем наборы группирования, которые представляют собой совокупность атрибутов для объединения данных. В этой главе описываются методики по извлечению нескольких наборов группирования в одном и том же запросе.

Представленный ниже материал является углубленным для тех, кто только начинает знакомиться с языком T-SQL; поэтому данные темы не являются обязательными к прочтению. Вы можете попробовать свои силы, если у вас не возникало никаких проблем с вышеизложенной информацией. В противном случае главу лучше пропустить и вернуться к ней, набравшись опыта.

## Оконные функции

**Оконными** называют функции, которые вычисляют скалярное выражение на основе некоего подмножества строк (окна), относящегося к текущей строке. Объявление оконных функций выполняется с помощью инструкции под названием `OVER`.

Если такое определение звучит для вас слишком формально, представьте, что речь идет о вычислении какого-то единого значения с учетом содержимого определенного набора записей. Классическим примером могут служить агрегатные (`SUM`, `COUNT` и `AVG`) и ранжирующие функции. Если вы читаете эту главу, у вас уже должен быть опыт выполнения подобных вычислений, в частности, вам уже знакомы группирующие и вложенные запросы. Однако эти методики порождают проблемы, которые легко обойти с помощью оконных функций.

Группирующие запросы могут дать общее представление о той теме, которую мы здесь рассматриваем, однако главное, как всегда, кроется в деталях. После группирования строк все вычисления должны производиться в контексте полученных групп, что не позволяет работать одновременно с отдельной строкой и агрегатом. Оконные функции лишены этого ограничения. Используя инструкцию `OVER` можно определить набор строк для обработки, который не будет иметь никакой связи с внешним запросом. Другими словами, группирующие запросы определяют наборы (группы), в рамках которых выполняются все вычисления — как внутренние, так и внешние. В оконных функциях контекст этих наборов не распространяется на внешний код.

Что касается вложенных запросов, то они тоже позволяют применять вычисления к набору строк, но при этом все операции выполняются только с исходными данными. К примеру, если ваш код содержит табличные операторы или фильтры, вы не сможете передать полученные результаты во вложенный запрос — вам придется заново повторять все вычисления. Оконные же функции могут работать с результирующим набором исходного запроса — для этого им не нужно повторно извлекать уже имеющиеся данные. Таким образом, все, что выполняется во внешнем запросе, автоматически влияет на каждую оконную функцию, которая в нем используется. Затем, используя разные элементы в инструкции `OVER`, можно ограничить подмножество строк, взятых из исходного результирующего набора.

Кроме того, оконные функции позволяют упорядочить результаты вычислений (в тех случаях, когда это уместно), не конфликтуя с реляционными аспектами результирующего набора. Порядок следования строк устанавливается на этапе вычисления, а не при выводе результата. При этом инструкция `ORDER BY` не гарантирует, что строки в результирующем наборе будут размещаться каким-то определенным образом. Если вы все же решите применить эту инструкцию для вывода данных, порядок следования строк может отличаться от того, который обеспечивает оконная функция.

Запрос, представленный ниже, обращается к представлению `Sales.EmpOrders`, которое использует оконную агрегатную функцию для вычисления общих текущих значений для каждого сотрудника и за каждый месяц.

```
USE TSQL2012;

SELECT empid, ordermonth, val,
       SUM(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                              AND CURRENT ROW) AS runval
FROM Sales.EmpOrders;
```

Результат, представленный в сокращенном виде, выглядит следующим образом.

empid	ordermonth	val	runval
1	2006-07-01	1614.88	1614.88
1	2006-08-01	5555.90	7170.78
1	2006-09-01	6651.00	13821.78
1	2006-10-01	3933.18	17754.96
1	2006-11-01	9562.65	27317.61
...			
2	2006-07-01	1176.00	1176.00
2	2006-08-01	1814.00	2990.00
2	2006-09-01	2950.80	5940.80
2	2006-10-01	5164.00	11104.80
2	2006-11-01	4614.58	15719.38
...			

(строк обработано: 192)

Определение окна содержится в инструкции `OVER` и состоит из трех главных частей: секционирования, упорядочения и рамок. Если оставить инструкцию `OVER ( )` пустой, функция получит окно, состоящее из строк исходного результирующего набора. Все, что вы добавляете в определение окна, сужает этот набор.

Оконная инструкция для секционирования (`PARTITION BY`) извлекает из исходного набора подмножество строк, секционные столбцы которых содержат те же значения, что и текущая строка. В вышеприведенном примере окно разделяется на секции на основе атрибута `empid`. Допустим, у нас есть строка, в которой этот атрибут равен 1; тогда для нее функция получит набор записей, в которых `empid` содержит значение 1.

Для упорядочения содержимого окна используется инструкция `ORDER BY`; она позволяет устанавливать рамки подмножества строк и не имеет ничего общего с сортировкой конечного результата. В нашем примере окно упорядочивается по столбцу `ordermonth`.

Имея упорядоченное окно, можно определить его рамки — два ограничителя, между которыми находится подмножество строк; для этого предусмотрена инструкция вида `ROWS BETWEEN <верхний предел> AND <нижний предел>`. В нашем примере рамки установлены между началом секционирования (`UNBOUNDED PRECEDING`) и текущей строкой (`CURRENT ROW`). Вместо `ROWS` можно использовать параметр `RANGE`, однако в Microsoft SQL Server 2012 он реализован в урезанном виде.

В итоге оконная функция возвращает общие текущие значения для каждого сотрудника с разбивкой по месяцам.

Обратите внимание: оконная функция работает с результирующим набором исходного запроса, который генерируется только на этапе `SELECT`. Следовательно, их можно размещать исключительно внутри инструкций `SELECT` и `ORDER BY`. Если у вас возникнет необходимость сослаться на оконную функцию на более раннем этапе (таком как `WHERE`), вам придется использовать табличное выражение. В этом случае определению функции, размещенному в списке `SELECT` во внутреннем запросе, присваивается псевдоним, на который потом можно будет ссылаться в любой части внешнего кода.

Для осмысления любой новой идеи нужно какое-то время, и оконные функции не являются исключением. Поработав с ними немного, вы поймете, что они хорошо вписываются в наши представления о вычислениях, а в долгосрочной перспективе позволят формулировать запросы в более естественной и понятной манере. Кроме того, в стандартных ситуациях оконные функции демонстрируют крайне высокую производительность.

Внедрение оконных функций в SQL Server проходило в два этапа. Их частичная реализация (только секционирование, без упорядочения и рамок) появилась в SQL Server 2005; в той же версии была объявлена полноценная поддержка ранжирующих функций (секционирование и упорядочение). С выходом SQL Server 2012 было добавлено множество возможностей, включая упорядочение и рамки для агрегатов, а также новые разновидности функций, которые выполняют смещение и распределение. И хотя в SQL Server до сих пор нет полной поддержки стандартной спецификации оконных функций, очень надеюсь, что компания Microsoft продолжит работу в этом направлении.

Более подробно о ранжировании, смещении и агрегировании мы поговорим в следующих разделах. Поскольку данная книга посвящена основам, некоторые вопросы останутся без внимания. Это касается оптимизации оконных функций, операций распределения и параметра `RANGE`, с помощью которого устанавливаются рамки окна.



## ДОПОЛНИТЕЛЬНЫЙ МАТЕРИАЛ

Поскольку оконные функции являются крайне универсальным и полезным инструментом, я посвятил им отдельную книгу под названием «Microsoft SQL Server 2012. Высокопроизводительный код T-SQL. Оконные функции» (русская редакция «БХВ-Петербург»). В ней раскрываются вопросы, связанные с оптимизацией и приводится множество примеров практического применения.

## Ранжирующие функции

Оконные ранжирующие функции позволяют оценивать каждую строку относительно других, используя различные приемы. SQL Server поддерживает четыре вида таких функций: ROW\_NUMBER, RANK, DENSE\_RANK и NTILE. Принцип их работы продемонстрирован в следующем примере.

```
SELECT orderid, custid, val,
       ROW_NUMBER() OVER(ORDER BY val) AS rownum,
       RANK() OVER(ORDER BY val) AS rank,
       DENSE_RANK() OVER(ORDER BY val) AS dr,
       NTILE(10) OVER(ORDER BY val) AS ntile
FROM Sales.OrderValues
ORDER BY val;
```

Результат, который генерирует этот запрос, показан ниже.

orderid	custid	val	rownum	rank	dr	ntile
10782	12	12.50	1	1	1	1
10807	27	18.40	2	2	2	1
10586	66	23.80	3	3	3	1
10767	76	28.00	4	4	4	1
10898	54	30.00	5	5	5	1
10900	88	33.75	6	6	6	1
10883	48	36.00	7	7	7	1
11051	41	36.00	8	7	7	1
10815	71	40.00	9	9	8	1
10674	38	45.00	10	10	9	1
...						
10691	63	10164.80	821	821	786	10
10540	63	10191.70	822	822	787	10
10479	65	10495.60	823	823	788	10
10897	37	10835.24	824	824	789	10
10817	39	10952.85	825	825	790	10
10417	73	11188.40	826	826	791	10
10889	65	11380.00	827	827	792	10
11030	71	12615.05	828	828	793	10
10981	34	15810.00	829	829	794	10
10865	63	16387.50	830	830	795	10

(строк обработано: 830)

Я уже описывал функцию ROW\_NUMBER в главе 2, но чтобы ничего не упустить, мы рассмотрим ее еще раз. Она присваивает строкам результирующего набора последовательные номера, исходя из логического порядка, который определяется вложенной

инструкцией `ORDER BY` внутри предложения `OVER`. Конечный результат запроса упорядочивается по столбцу `val`; следовательно, значения результирующего набора растут вместе с номерами строк. Хотя номера увеличиваются в любом случае, даже если значения, по которым выполняется сортировка, оказываются одинаковыми. Таким образом, если во вложенной инструкции `ORDER BY` указать неуникальные элементы, как в предыдущем примере, запрос получится недетерминированным (то есть можно будет получить несколько корректных результатов). Представьте, что две строки со значением 36 имеют порядковые номера 7 и 8. Любое их размещение друг относительно друга будет считаться корректным. Чтобы нумерация была детерминированной, вам придется сделать список `ORDER BY` уникальным, добавив в него новые элементы; это позволит однозначно идентифицировать каждую строку. В качестве такого дополнительного элемента может выступить, например, столбец `orderid`.

Как уже упоминалось ранее, функция `ROW_NUMBER` должна генерировать уникальные значения даже при наличии равноценных сортировочных значений. Если вы хотите, чтобы совпадающие значения отражались на ранжировании, лучше воспользоваться функциями `RANK` или `DENSE_RANK`. Они похожи на `ROW_NUMBER`, но в случае совпадения значений, по которым упорядочивается набор, допускают одинаковые порядковые номера. При этом функция `DENSE_RANK`, в отличие от `RANK`, вычисляет степень ранжирования только на основе уникальных сортировочных значений. Например, если бы в предыдущем запросе функция `RANK` вернула 9, это означало бы, что существует восемь строк с меньшими значениями. Функция `DENSE_RANK` в той же ситуации указывала бы на восемь уникальных значений меньше текущего.

Функция `NTILE` позволяет связывать строки результирующего набора с группами строк одинакового размера. Количество групп указывается в качестве входящего аргумента, а с помощью инструкции `OVER` определяется логический порядок следования строк. Запрос, приведенный в предыдущем примере, возвращает 830 записей, при этом мы запросили 10 групп; следовательно, размер группы равен 83 (830 разделить на 10). Сортировка выполняется по столбцу `val`. Это означает, что 83 строки с самым меньшим значением были собраны в группу под номером 1, следующие 83 строки попали в группу 2 и т. д. По своему принципу работы функция `NTILE` похожа на `ROW_NUMBER`. Она тоже присваивает порядковые номера, но не отдельным строкам, а целым группам, размер которых определяется входящим аргументом. Остаток, полученный в результате деления количества строк на количество групп, равномерно распределяется между группами, которые находятся вначале. Например, если бы у нас было 102 строки и пять групп, первые две группы состояли бы не из 20, а из 21 строки.

Функции ранжирования поддерживают инструкции для секционирования окон. Как вы помните, операция секционирования оставляет в окне только те записи, у которых значения соответствующих атрибутов такие же, как у текущей строки. Например, выражение `ROW_NUMBER() OVER(PARTITION BY custid ORDER BY val)` вместо всего результирующего набора нумерует отдельные подмножества строк с одинаковым значением `custid`. Давайте рассмотрим это выражение в контексте реального запроса.

```
SELECT orderid, custid, val,
       ROW_NUMBER() OVER(PARTITION BY custid
                          ORDER BY val) AS rownum
FROM Sales.OrderValues
ORDER BY custid, val;
```

Результат (в сокращенном виде) представлен ниже.

orderid	custid	val	rownum
10702	1	330.00	1
10952	1	471.20	2
10643	1	814.50	3
10835	1	845.80	4
10692	1	878.00	5
11011	1	933.50	6
10308	2	88.80	1
10759	2	320.00	2
10625	2	479.75	3
10926	2	514.40	4
10682	3	375.50	1
...			

(строк обработано: 830)

Как видите, номера строк вычисляются отдельно для каждого клиента; все выглядит так, как будто они сбрасываются для каждой новой группы.

Запомните, сортировка строк внутри окна не имеет ничего общего с выводом конечного результата и не влияет на его реляционную сущность. Чтобы упорядочить результирующий набор, используйте во внешнем запросе инструкцию `ORDER BY`, как было показано в предыдущих двух примерах с ранжирующими функциями.

Как вы уже знаете из главы 2, оконные функции вызываются как часть выражения, размещенного в списке `SELECT`; это происходит до вызова инструкции `DISTINCT`. Чтобы понять важность данного факта, приведу конкретный пример. Сейчас представление `OrderValues` возвращает 830 строк с 795 уникальными значениями в столбце `val`. Теперь посмотрите на запрос, приведенный ниже, и на результат его выполнения.

```
SELECT DISTINCT val, ROW_NUMBER() OVER(ORDER BY val) AS rownum
FROM Sales.OrderValues;
```

val	rownum
12.50	1
18.40	2
23.80	3
28.00	4
30.00	5
33.75	6
36.00	7
36.00	8
40.00	9
45.00	10
...	
12615.05	828
15810.00	829
16387.50	830

(строк обработано: 830)

Функция `ROW_NUMBER` обрабатывается перед инструкцией `DISTINCT`. Сначала каждой из 830 строк, полученных из представления `OrderValues`, назначаются уни-

кальные номера. Затем выполняется инструкция `DISTINCT`, которая должна удалять дубликаты. Но ее использование теряет всякий смысл, так как мы уже гарантировали уникальность всех строк. Поэтому в сочетании с функцией `ROW_NUMBER` ее лучше не применять. Если вы хотите пронумеровать 795 уникальных строк, вам нужно использовать другой подход. Например, вы могли бы написать следующий запрос, поскольку инструкция `GROUP BY` выполняется перед этапом `SELECT`.

```
SELECT val, ROW_NUMBER() OVER(ORDER BY val) AS rownum
FROM Sales.OrderValues
GROUP BY val;
```

Ниже представлен результат (в сокращенном виде).

val	rownum
-----	-----
12.50	1
18.40	2
23.80	3
28.00	4
30.00	5
33.75	6
36.00	7
40.00	8
45.00	9
48.00	10
...	
12615.05	793
15810.00	794
16387.50	795

(строк обработано: 795)

Здесь на этапе `GROUP BY` генерируется 795 групп для такого же количества уникальных значений, затем инструкция `SELECT` возвращает для каждой группы по одной строке с дополнительным столбцом `val`, в котором хранится уникальный номер.

## Функции со смещением

Оконные функции со смещением позволяют получить элемент строки, которая находится на определенном расстоянии от текущей записи или границ окна. SQL Server 2012 поддерживает четыре такие функции: `LAG`, `LEAD`, `FIRST_VALUE` и `LAST_VALUE`.

Функции `LAG` и `LEAD` поддерживают оконные инструкции для секционирования и упорядочения, однако ничего похожего на границы в них нет. Они позволяют получить элемент строки, которая находится на каком-то расстоянии от текущей записи в рамках секции, упорядоченной определенным образом. Функция `LAG` выполняет смещение назад, а `LEAD` — вперед. В качестве первого обязательного аргумента выступает элемент, который вы хотите вернуть; второй и третий аргументы являются опциональными и представляют соответственно смещение (по умолчанию равно 1) и значение, которое возвращается в случае отсутствия запрашиваемой строки (по умолчанию `NULL`).

В качестве примера рассмотрим запрос, который извлекает информацию о заказах из представления `OrderValues`. К каждой строке применяются функции



LAG и LEAD, которые возвращают значения предыдущего и следующего заказов соответственно.

```
SELECT custid,orderid, val,
       LAG(val) OVER(PARTITION BY custid
                     ORDER BY orderdate,orderid) AS prevval,
       LEAD(val) OVER(PARTITION BY custid
                      ORDER BY orderdate,orderid) AS nextval
FROM Sales.OrderValues;
```

Вот результат этого запроса, представленный в сокращенном виде.

custid	orderid	val	prevval	nextval
1	10643	814.50	NULL	878.00
1	10692	878.00	814.50	330.00
1	10702	330.00	878.00	845.80
1	10835	845.80	330.00	471.20
1	10952	471.20	845.80	933.50
1	11011	933.50	471.20	NULL
2	10308	88.80	NULL	479.75
2	10625	479.75	88.80	320.00
2	10759	320.00	479.75	514.40
2	10926	514.40	320.00	NULL
3	10365	403.20	NULL	749.06
3	10507	749.06	403.20	1940.85
3	10535	1940.85	749.06	2082.00
3	10573	2082.00	1940.85	813.37
3	10677	813.37	2082.00	375.50
3	10682	375.50	813.37	660.00
3	10856	660.00	375.50	NULL
...				

(строк обработано: 830)

Мы не указали смещение, поэтому функции исходят из того, что оно равно 1; другими словами, они возвращают значения заказов, которые размещены непосредственно перед текущей строкой и сразу после нее. Нами также был опущен третий аргумент, поэтому при отсутствии следующей или предыдущей строки возвращается NULL. Выражение LAG(val, 3, 0) вернуло бы значение, которое находится в трех строках позади, а если бы такового не оказалось, мы получили бы 0.

В этом запросе мы просто извлекли значения из предыдущего и следующего заказов, но обычно данные, полученные таким образом, участвуют в последующих вычислениях. Например, вы можете посчитать разницу между значениями текущего и предыдущего/следующего заказов: val - LAG(val) OVER(...) или val - LEAD(val) OVER(...).

Функции FIRST\_VALUE и LAST\_VALUE позволяют извлекать элементы из первой и последней строк в рамках одного окна. Они поддерживают оконные инструкции для секционирования, упорядочения и определения границ. Чтобы получить элемент из первой строки, используйте функцию FIRST\_VALUE с рамками ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Элемент из последней строки извлекается с помощью LAST\_VALUE и рамок ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING. Стоит отметить, что если вы укажете инструкцию ORDER BY без определения границ секции (например, ROWS), то нижняя

граница по умолчанию будет равна `CURRENT ROW`, что совершенно не подходит в случае использования функции `LAST_VALUE`. Границы секции должны быть явно определены даже для функции `FIRST_VALUE`; причины этого требования выходят за рамки тем, затрагиваемых в данной книге.

В качестве примера рассмотрим следующий запрос, который с помощью функций `FIRST_VALUE` и `LAST_VALUE` возвращает значения из первого и последнего заказов соответственно.

```
SELECT custid,orderid, val,
       FIRST_VALUE(val) OVER(PARTITION BY custid
                             ORDER BY orderdate,orderid
                             ROWS BETWEEN UNBOUNDED PRECEDING
                                     AND CURRENT ROW) AS firstval,
       LAST_VALUE(val)  OVER(PARTITION BY custid
                             ORDER BY orderdate,orderid
                             ROWS BETWEEN CURRENT ROW
                                     AND UNBOUNDED FOLLOWING) AS lastval
FROM Sales.OrderValues
ORDER BY custid,orderdate,orderid;
```

Результат, генерируемый этим запросом, показан ниже (в сокращенном виде).

custid	orderid	val	firstval	lastval
1	10643	814.50	814.50	933.50
1	10692	878.00	814.50	933.50
1	10702	330.00	814.50	933.50
1	10835	845.80	814.50	933.50
1	10952	471.20	814.50	933.50
1	11011	933.50	814.50	933.50
2	10308	88.80	88.80	514.40
2	10625	479.75	88.80	514.40
2	10759	320.00	88.80	514.40
2	10926	514.40	88.80	514.40
3	10365	403.20	403.20	660.00
3	10507	749.06	403.20	660.00
3	10535	1940.85	403.20	660.00
3	10573	2082.00	403.20	660.00
3	10677	813.37	403.20	660.00
3	10682	375.50	403.20	660.00
3	10856	660.00	403.20	660.00
...				

(строк обработано: 830)

Как и в случае с функциями `LAG` и `LEAD`, полученные значения обычно используются в дальнейших вычислениях. Например, вы могли бы посчитать разницу между значениями текущей и первой/последней строками: `val - FIRST_VALUE(val) OVER(...)` или `val - LAST_VALUE(val) OVER(...)`.

## Агрегатные функции

В предыдущих версиях SQL Server агрегатные оконные функции поддерживали только инструкцию для секционирования. Но с выходом SQL Server 2012, в котором появилась реализация упорядочения и определения рамок, спектр их применения значительно расширился.

Начнем с примера, в котором используется только секционирование. Как вы помните, в сочетании с пустыми скобками инструкция `OVER` возвращает в функцию окно со всеми строками, извлеченными из исходного запроса — выражение `SUM(val) OVER()` вернет общую сумму всех значений. Если добавить инструкцию `PARTITION BY`, функция получит ограниченное окно со строками, у которых значения элементов секционирования совпадают с аналогичными элементами в текущей записи. К примеру, выражение `SUM(val) OVER(PARTITION BY custid)` вернет сумму значений для текущего клиента.

Ниже представлен запрос к представлению `OrderValues`, который в каждой строке возвращает сумму всех заказов и общую сумму заказов для отдельных клиентов.

```
SELECTorderid, custid, val,
       SUM(val) OVER() AS totalvalue,
       SUM(val) OVER(PARTITION BY custid) AS custtotalvalue
FROM Sales.OrderValues;
```

Результат, генерируемый этим запросом, будет следующим.

orderid	custid	val	totalvalue	custtotalvalue
10643	1	814.50	1265793.22	4273.00
10692	1	878.00	1265793.22	4273.00
10702	1	330.00	1265793.22	4273.00
10835	1	845.80	1265793.22	4273.00
10952	1	471.20	1265793.22	4273.00
11011	1	933.50	1265793.22	4273.00
10926	2	514.40	1265793.22	1402.95
10759	2	320.00	1265793.22	1402.95
10625	2	479.75	1265793.22	1402.95
10308	2	88.80	1265793.22	1402.95
10365	3	403.20	1265793.22	7023.98
...				

(строка обработано: 830)

В столбце `totalvalue` выводится общая цена, вычисленная на основе всех строк. Столбец `custtotalvalue` хранит общую цену для всех строк, у которых значение `custid` совпадает с текущей записью.

Как уже упоминалось ранее, одно из преимуществ оконных функций заключается в том, что благодаря возможности агрегировать полученные элементы в одной строке они позволяют создавать запросы, которые комбинируют сами данные и их агрегаты. Например, следующий запрос вычисляет для каждой строки долю, которую текущее значение занимает в общей сумме и в сумме заказов для отдельного клиента.

```
SELECTorderid, custid, val,
       100. * val / SUM(val) OVER() AS pctall,
       100. * val / SUM(val) OVER(PARTITION BY custid) AS pctcust
FROM Sales.OrderValues;
```

Ниже представлен результат (в сокращенном виде).

orderid	custid	val	pctall	pctcust
10643	1	814.50	0.0643470029014691672941	19.0615492628130119354083
10692	1	878.00	0.0693636200705830925528	20.5476246197051252047741

10702	1	330.00	0.0260706089103558320528	7.7229113035338169904048
10835	1	845.80	0.0668197606556938265161	19.7940556985724315469225
10952	1	471.20	0.0372256694501808123130	11.0273812309852562602387
11011	1	933.50	0.0737482224782338461253	21.8464778843903580622513
10926	2	514.40	0.0406385491620819394181	36.6655974910011048148544
10759	2	320.00	0.0252805904585268674452	22.8090808653195053280587
10625	2	479.75	0.0379011352264945770526	34.1958017035532271285505
10308	2	88.80	0.0070153638522412057160	6.3295199401261627285362
10365	3	403.20	0.0318535439777438529809	5.7403352515240647040566

...

(строк обработано: 830)

С выходом SQL Server 2012 агрегатные функции начали поддерживать упорядочение и задание границ. Это дает возможность производить более сложные вычисления, например промежуточные и скользящие агрегаты, определение периода, прошедшего с начала года, и т. д. Давайте еще раз рассмотрим пример, который я использовал в начале раздела об оконных функциях.

```
SELECT empid, ordermonth, val,
       SUM(val) OVER(PARTITION BY empid
                     ORDER BY ordermonth
                     ROWS BETWEEN UNBOUNDED PRECEDING
                              AND CURRENT ROW) AS runval
FROM Sales.EmpOrders;
```

Этот запрос генерирует следующий набор (представленный в сокращенном виде).

empid	ordermonth	val	runval
1	2006-07-01	1614.88	1614.88
1	2006-08-01	5555.90	7170.78
1	2006-09-01	6651.00	13821.78
1	2006-10-01	3933.18	17754.96
1	2006-11-01	9562.65	27317.61
...			
2	2006-07-01	1176.00	1176.00
2	2006-08-01	1814.00	2990.00
2	2006-09-01	2950.80	5940.80
2	2006-10-01	5164.00	11104.80
2	2006-11-01	4614.58	15719.38
...			

...

(строк обработано: 192)

Каждая строка в представлении EmpOrders хранит информацию о дате выполнения заказа и сотруднике, который этот заказ обрабатывал. Запрос возвращает для каждой записи месячную сумму и общую промежуточную сумму всех заказов сотрудника с момента начала его активности и по текущий месяц. Чтобы выполнить вычисление для каждого сотрудника отдельно, необходимо секционировать окно по столбцу empid. Затем нужно упорядочить строки на основе столбца ordermonth и определить границы окна: ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. Полученные рамки включают информацию о «всех заказах, находящихся между началом секции и текущим месяцем».

Обозначать границы окна в SQL Server можно разными способами, например посредством смещения вперед/назад относительно текущей строки. Чтобы захватить все

элементы, начиная второй строкой сзади и заканчивая первой строкой спереди, можно воспользоваться выражением `ROWS BETWEEN 2 PRECEDING AND 1 FOLLOWING`. Если же вы хотите снять верхнее ограничение, укажите `UNBOUNDED FOLLOWING`. Определять границы можно и посредством ключевого слова `RANGE`, однако в SQL Server оно имеет довольно ограниченное применение и не будет здесь рассматриваться. Желательно воздержаться от использования этого варианта ввиду его несовершенной реализации.

Оконные функции обладают широкими возможностями и могут использоваться в разных ситуациях. Время и силы, потраченные на их изучение, окупят себя сторицей.

## Разворачивание данных

**Разворачивание данных** подразумевает преобразование строк в столбцы с возможным попутным агрегированием значений. Не волнуйтесь, если данное определение кажется вам не совсем понятным; изучение этой темы лучше начинать с рассмотрения конкретных примеров. Часто разворачивание данных происходит на этапе их вывода. Здесь мы сосредоточимся на тех случаях, когда этот процесс выполняется с помощью языка T-SQL внутри самой БД.

Во всех примерах, которые будут рассматриваться в текущей главе, используется таблица `dbo.Orders`, которую легко создать и заполнить с помощью кода, приведенного в листинге 7.1.

### Листинг 7.1. Код для создания и заполнения таблицы `dbo.Orders`

```
USE TSQL2012;

IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;

CREATE TABLE dbo.Orders
(
   orderid    INT           NOT NULL,
    orderdate DATE          NOT NULL,
    empid     INT           NOT NULL,
    custid    VARCHAR(5)    NOT NULL,
    qty       INT           NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);

INSERT INTO dbo.Orders(orderid, orderdate, empid, custid, qty)
VALUES
    (30001, '20070802', 3, 'A', 10),
    (10001, '20071224', 2, 'A', 12),
    (10005, '20071224', 1, 'B', 20),
    (40001, '20080109', 2, 'A', 40),
    (10006, '20080118', 1, 'C', 14),
    (20001, '20080212', 2, 'B', 12),
    (40005, '20090212', 3, 'A', 10),
    (20002, '20090216', 1, 'C', 20),
    (30003, '20090418', 2, 'B', 15),
    (30004, '20070418', 3, 'C', 22),
    (30007, '20090907', 3, 'D', 30);

SELECT * FROM dbo.Orders;
```

Запрос, размещенный в конце этого листинга, выводит содержимое таблицы `dbo.Orders`, как показано ниже.

orderid	orderdate	empid	custid	qty
10001	2007-12-24	2	A	12
10005	2007-12-24	1	B	20
10006	2008-01-18	1	C	14
20001	2008-02-12	2	B	12
20002	2009-02-16	1	C	20
30001	2007-08-02	3	A	10
30003	2009-04-18	2	B	15
30004	2007-04-18	3	C	22
30007	2009-09-07	3	D	30
40001	2008-01-09	2	A	40
40005	2009-02-12	3	A	10

Прежде чем дальше рассматривать процесс разворачивания данных, давайте попробуем получить общее количество заказанного товара для каждого сотрудника и клиента. Для этого достаточно выполнить простой запрос.

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid, custid;
```

Результат будет следующим.

empid	custid	sumqty
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30

А теперь представьте, что вам необходимо вывести этот результат так, как показано в таблице 7.1.

**Таблица 7.1.** Развернутое представление общего количество заказанного товара для сотрудников (строки) и клиентов (столбцы)

empid	A	B	C	D
1	NULL	20	34	NULL
2	52	NULL	22	NULL
3	20	NULL	22	30

Здесь вы можете видеть агрегированные и развернутые данные из таблицы `dbo.Orders`; процесс, в результате которого они были представлены таким образом, называется **разворачиванием**.

Процесс разворачивания состоит из трех логических этапов, с каждым из которых связан соответствующий элемент: группирование (по строкам), распределение (по столбцам) и агрегирование (с помощью агрегатной функции).

В этом примере для каждого уникального идентификатора сотрудника нужно сгенерировать по одной строке. Это означает, что строки в таблице `dbo.Orders` должны быть сгруппированы по атрибуту `empid`, который в данном случае выступает группирующим элементом.

В таблице `dbo.Orders` есть два столбца, которые хранят соответственно идентификаторы клиентов и количество товаров, которое эти клиенты заказывали. Процесс разворачивания предполагает получение разных результирующих столбцов для каждого уникального идентификатора; при этом каждый столбец должен содержать общее количество заказанного товара для соответствующего клиента. Фактически это «распределение» количества товара по атрибуту `custid`.

Процесс разворачивания включает этап группирования, поэтому вам необходимо агрегировать данные, чтобы сгенерировать итоговый результат на «пересечении» группирующего и распределяющего элементов. Для этого нужно определить агрегатную функцию (в нашем случае `SUM`) и агрегатный элемент (в данном примере атрибут `qty`).

Еще раз: процесс разворачивания состоит из группирования, распределения и агрегирования. В данном примере мы группируем по атрибуту `empid`, распределяем (количество товара) по `custid` и агрегируем с помощью функции `SUM(qty)`. После определения всех этих элементов останется только расставить их в нужных местах внутри универсального шаблона для разворачивания данных.

В этой главе рассматривается два подхода к процессу разворачивания; один является стандартным, а другой поддерживается только в языке T-SQL (с использованием оператора `PIVOT`).

## Стандартные средства

Стандартное решение для разворачивания данных включает все три этапа, которые выполняются в довольно прозрачном стиле.

Группирование производится за счет инструкции `GROUP BY`; в нашем случае это будет выглядеть как `GROUP BY empid`.

Этап распределения реализуется посредством инструкции `SELECT` и выражения `CASE`, которые относятся к каждому столбцу. Значения распределяющего элемента нужно знать заранее, чтобы указать для каждого из них отдельное выражение. В нашем случае таких выражений будет четыре — по одному на каждого клиента (A, B, C и D). Например, вот как выглядит код для клиента A.

```
CASE WHEN custid = 'A' THEN qty END
```

Это выражение возвращает количество заказанного товара, но только если заказ делал клиент A; в противном случае возвращается `NULL`. Как вы помните, если в выражении `CASE` нет инструкции `ELSE`, по умолчанию используется `ELSE NULL`. Это означает, что в целевом столбце будут появляться значения, связанные только с клиентом A, а во всех остальных случаях — применяться отметка `NULL`.

Если вы не знаете заранее, какие значения вам нужно распределять (в нашем случае это отдельные идентификаторы клиентов), и хотите получить их из БД, следует использовать динамические конструкции языка SQL, чтобы на лету создать и выполнить строку с запросом. Динамическое разворачивание данных рассматривается в главе 10.

Наконец, этап агрегации достигается за счет применения к результату, который возвращается каждым выражением CASE, соответствующей агрегатной функции (SUM, в нашем случае). Например, следующее выражение генерирует итоговый столбец для клиента A.

```
SUM(CASE WHEN custid = 'A' THEN qty END) AS A
```

Конечно, выбор агрегатной функции зависит от конкретного запроса; это могут быть функции MAX, MIN или COUNT.

Ниже представлен полноценный запрос, который разворачивает данные о заказах, возвращая общее количество товара для каждого сотрудника (по строкам) и клиента (по столбцам).

```
SELECT empid,
       SUM(CASE WHEN custid = 'A' THEN qty END) AS A,
       SUM(CASE WHEN custid = 'B' THEN qty END) AS B,
       SUM(CASE WHEN custid = 'C' THEN qty END) AS C,
       SUM(CASE WHEN custid = 'D' THEN qty END) AS D
FROM   dbo.Orders
GROUP BY empid;
```

Результат, который генерирует этот запрос, показан в таблице 7.1.

## Встроенный оператор PIVOT

Язык T-SQL поддерживает нестандартный оператор под названием PIVOT. Как и любые другие табличные операторы (например, JOIN), он выполняется в контексте инструкции FROM. Он берет исходную таблицу или табличное выражение, разворачивает данные и возвращает итоговый результат. Оператор PIVOT состоит из тех же логических этапов обработки, которые мы рассматривали ранее (группирование, распределение и агрегирование), и с теми же элементами. Разница заключается в том, что он использует синтаксис, встроенный в язык T-SQL.

В целом, запрос с оператором PIVOT выглядит следующим образом.

```
SELECT ...
FROM <таблица_или_табличное_выражение>
     PIVOT (<агрегатная_функция> (<агрегатный_элемент>)
           FOR <распределяющий_элемент>
           IN (<целевые_столбцы>)) AS <псевдоним_итоговой_таблицы>
...;
```

В скобках оператора PIVOT нужно указать агрегатную функцию (в нашем примере это SUM), агрегатный элемент (qty), распределяющий элемент (custid) и список целевых столбцов (A, B, C, D). После этого должен быть назначен псевдоним для итоговой таблицы.

Важно отметить, что для оператора PIVOT не нужно указывать отдельный группирующий элемент, в результате чего исчезает необходимость в инструкции GROUP BY. Группирование выполняется автоматически на основе всех атрибутов исходной таблицы (или табличного выражения), которые не были указаны для распределения или агрегации. Вы должны убедиться в том, что



ваша таблица не содержит атрибутов, которые не используются на данных трех этапах, чтобы группирование происходило только по нужным элементам. Для этого оператор `PIVOT` можно применять не к самой таблице (в нашем случае `Orders`), а к табличному выражению, в котором содержатся лишь те атрибуты, которые участвуют в процессе разворачивания данных. Вот как будет выглядеть наш изначальный пример, если его переписать с использованием оператора `PIVOT`.

```
SELECT empid, A, B, C, D
FROM (SELECT empid, custid, qty
      FROM dbo.Orders) AS D
     PIVOT(SUM(qty) FOR custid IN(A, B, C, D)) AS P;
```

Вместо того чтобы обращаться напрямую к таблице `dbo.Orders`, оператор `PIVOT` работает с производной таблицей под названием `D`, которая содержит только те элементы, которые участвуют в разворачивании, — `empid`, `custid` и `qty`. Если отбросить распределяющий (`custid`) и агрегирующий (`qty`) атрибуты, останется атрибут `empid`, который и будет использоваться для группирования.

Результат выполнения этого запроса вы можете наблюдать в таблице 7.1.

Чтобы понять, зачем нам здесь понадобилось табличное выражение, давайте попробуем применить оператор `PIVOT` непосредственно к таблице `dbo.Order`.

```
SELECT empid, A, B, C, D
FROM dbo.Orders
     PIVOT(SUM(qty) FOR custid IN(A, B, C, D)) AS P;
```

Таблица `dbo.Orders` содержит столбцы `orderid`, `orderdate`, `empid`, `custid` и `qty`. Если отбросить элементы для распределения и агрегирования (`custid` и `qty`), останутся атрибуты `orderid`, `orderdate` и `empid`, которые будут использоваться при группировании. Таким образом, этот запрос вернет следующий результат.

empid	A	B	C	D
2	12	NULL	NULL	NULL
1	NULL	20	NULL	NULL
1	NULL	NULL	14	NULL
2	NULL	12	NULL	NULL
1	NULL	NULL	20	NULL
3	10	NULL	NULL	NULL
2	NULL	15	NULL	NULL
3	NULL	NULL	22	NULL
3	NULL	NULL	NULL	30
2	40	NULL	NULL	NULL
3	10	NULL	NULL	NULL

(строк обработано: 11)

Поскольку столбец `orderid` является одним из группирующих элементов, вы получите по одной строке для каждого заказа, а не для каждого сотрудника. Если бы мы использовали стандартные средства для разворачивания данных, атрибуты `orderid`, `orderdate` и `empid` пришлось бы перечислить в инструкции `GROUP BY`, как показано далее.

```

SELECT empid,
       SUM(CASE WHEN custid = 'A' THEN qty END) AS A,
       SUM(CASE WHEN custid = 'B' THEN qty END) AS B,
       SUM(CASE WHEN custid = 'C' THEN qty END) AS C,
       SUM(CASE WHEN custid = 'D' THEN qty END) AS D
FROM dbo.Orders
GROUP BY orderid, orderdate, empid;

```

Настоятельно не рекомендую вам работать с исходной таблицей напрямую, даже если она не содержит лишних атрибутов. Ведь в будущем в нее могут быть добавлены новые столбцы, что сделает результаты ваших запросов некорректными. Вместо этого в качестве входящих аргументов для оператора `PIVOT` желательно использовать табличные выражения; можно сказать, что это не менее важно, чем соблюдение синтаксиса.

Давайте рассмотрим еще один пример разворачивания данных. Ранее мы выводили сотрудников в строках и клиентов в столбцах, теперь же сделаем по-другому: группирующим элементом будет `custid`, распределяющим — `empid` и агрегирующим — все та же функция `SUM(qty)`. Теперь, когда вы знаете, как в целом выглядит разворачивание (стандартное или с использованием встроеного оператора), вам достаточно просто разместить элементы на подходящих позициях. В следующем запросе это делается с помощью встроеного оператора `PIVOT`.

```

SELECT custid, [1], [2], [3]
FROM (SELECT empid, custid, qty
      FROM dbo.Orders) AS D
     PIVOT(SUM(qty) FOR empid IN([1], [2], [3])) AS P;

```

В столбце `empid` исходной таблицы хранятся идентификаторы 1, 2 и 3, и в результирующем наборе они превращаются в имена атрибутов. Однако имена объектов в БД не могут начинаться с цифры, поэтому в выражении `PIVOT IN` вы должны выделить их с помощью квадратных скобок.

Этот запрос возвращает следующий результат.

custid	1	2	3
A	NULL	52	20
B	20	27	NULL
C	34	NULL	22
D	NULL	NULL	30

## Отмена разворачивания данных

**Отмена разворачивания данных** — это фактически преобразование столбцов в строки. Как правило, подобный процесс подразумевает работу с данными, которые ранее были развернуты; при этом из одной исходной строки генерируется несколько итоговых строк с уникальными значениями в одном из столбцов. Другими словами, каждая запись развернутой таблицы при обратном процессе может превратиться во множество записей — по одной на каждое значение в заданном столбце. Поначалу такое объяснение может показаться не совсем понятным, поэтому обратимся к конкретному примеру.

Запустите следующий код, чтобы создать и заполнить таблицу EmpCustOrders в БД TSQL2012.

```
USE TSQL2012;

IF OBJECT_ID('dbo.EmpCustOrders', 'U') IS NOT NULL DROP TABLE dbo.
EmpCustOrders;

CREATE TABLE dbo.EmpCustOrders
(
    empid INT NOT NULL
        CONSTRAINT PK_EmpCustOrders PRIMARY KEY,
    A VARCHAR(5) NULL,
    B VARCHAR(5) NULL,
    C VARCHAR(5) NULL,
    D VARCHAR(5) NULL
);

INSERT INTO dbo.EmpCustOrders(empid, A, B, C, D)
SELECT empid, A, B, C, D
FROM (SELECT empid, custid, qty
      FROM dbo.Orders) AS D
      PIVOT(SUM(qty) FOR custid IN(A, B, C, D)) AS P;

SELECT * FROM dbo.EmpCustOrders;
```

Содержимое этой таблицы показано ниже.

empid	A	B	C	D
1	NULL	20	34	NULL
2	52	27	NULL	NULL
3	20	NULL	22	30

У каждого сотрудника есть отдельная строка, а для клиентов выделено четыре столбца: A, B, C и D. Кроме того, на пересечении строк и столбцов можно наблюдать количество заказанного товара (и для клиентов, и для сотрудников). Обратите внимание, что пересечения, с которыми не связана информация о заказах, дают NULL. Представьте, что вам нужно отменить разворачивание данных, то есть сделать так, чтобы итоговые строки хранили данные о каждом сотруднике, клиенте и количестве заказанного товара. Результат будет выглядеть так.

empid	custid	qty
1	B	20
1	C	34
2	A	52
2	B	27
3	A	20
3	C	22
3	D	30

В следующих разделах мы рассмотрим два подхода к решению этой проблемы; один является стандартным, а другой поддерживается только в языке T-SQL (с использованием оператора UNPIVOT).

## Стандартные средства

Стандартный подход к отмене разворачивания данных явным образом реализует три логических этапа обработки: создание копий, извлечение элементов и удаление лишних пересечений.

Первым делом нужно создать несколько копий каждой исходной строки — их число определяется количеством столбцов, которые разворачиваются. В нашем случае для столбцов **A**, **B**, **C** и **D**, которые представляют идентификаторы клиентов, нужно сгенерировать по одной копии. В реляционной алгебре и языке SQL такая операция называется **декартовым произведением** (или перекрестным соединением). Для ее выполнения необходимо соединить таблицу **EmpCustOrders** и виртуальный набор, который содержит по одной строке для каждого клиента.

Чтобы создать его, можно воспользоваться конструктором табличных значений, представленным инструкцией **VALUES**. Запрос, реализующий первый этап нашего решения, выглядит так.

```
SELECT *
FROM dbo.EmpCustOrders
     CROSS JOIN (VALUES ('A'), ('B'), ('C'), ('D')) AS Custs(custid);
```

Если вы еще не знакомы с инструкцией **VALUES**, подробную информацию о ней можно найти в главе 8.

Запрос, который в нашем случае реализует первый этап, выводит следующий результат.

empid	A	B	C	D	custid
1	NULL	20	34	NULL	A
1	NULL	20	34	NULL	B
1	NULL	20	34	NULL	C
1	NULL	20	34	NULL	D
2	52	27	NULL	NULL	A
2	52	27	NULL	NULL	B
2	52	27	NULL	NULL	C
2	52	27	NULL	NULL	D
3	20	NULL	22	30	A
3	20	NULL	22	30	B
3	20	NULL	22	30	C
3	20	NULL	22	30	D

Как видите, мы получили четыре копии каждой исходной строки — по одной для клиентов **A**, **B**, **C** и **D**.

Второй этап заключается в создании столбца (в нашем случае это **qty**), который будет хранить значения, связанные с текущим клиентом. Например, если текущее значение столбца **custid** равно **A**, столбец должен вернуть содержимое атрибута под названием **A**. Этот алгоритм можно реализовать с помощью выражения **CASE**, как показано ниже.

```
SELECT empid, custid,
       CASE custid
         WHEN 'A' THEN A
```

```
        WHEN 'B' THEN B
        WHEN 'C' THEN C
        WHEN 'D' THEN D
    END AS qty
FROM dbo.EmpCustOrders
    CROSS JOIN (VALUES('A'),('B'),('C'),('D')) AS Custs(custid);
```

Этот запрос возвращает следующий результат.

empid	custid	qty
1	A	NULL
1	B	20
1	C	34
1	D	NULL
2	A	52
2	B	27
2	C	NULL
2	D	NULL
3	A	20
3	B	NULL
3	C	22
3	D	30

Как вы помните, для представления лишних пересечений в исходной таблице использовались отметки NULL. Чтобы от них избавиться, нужно определить табличное выражение, основанное на запросе, который реализует второй этап, затем отфильтровать значения NULL во внешнем коде. Итоговое решение выглядит следующим образом.

```
SELECT *
FROM (SELECT empid, custid,
    CASE custid
        WHEN 'A' THEN A
        WHEN 'B' THEN B
        WHEN 'C' THEN C
        WHEN 'D' THEN D
    END AS qty
    FROM dbo.EmpCustOrders
        CROSS JOIN (VALUES('A'),('B'),('C'),('D'))
            AS Custs(custid)) AS D

WHERE qty IS NOT NULL;
```

Ниже представлен результат.

empid	custid	qty
1	B	20
1	C	34
2	A	52
2	B	27
3	A	20
3	C	22
3	D	30

## Встроенный оператор UNPIVOT

Отмена разворачивания данных подразумевает сведение любой исходной таблицы к двум итоговым столбцам. В этом примере у нас есть столбцы **A**, **B**, **C** и **D**, из которых нужно получить атрибуты под названием `custid` и `qty`; первый будет хранить имена исходных столбцов (**A**, **B**, **C** и **D**), а во втором будут находиться их значения (в нашем случае это количество заказанного товара). В SQL Server имеется поддержка очень изящного и минималистического табличного оператора `UNPIVOT`, который обладает следующим синтаксисом.

```
SELECT ...
FROM <таблица_или_табличное_выражение>
    UNPIVOT(<целевой_столбец_для_хранения_исходных_значений>
        FOR <целевой_столбец_для_хранения_имен_исходных_столбцов>
            IN (<исходные_столбцы>)) AS <псевдоним_итоговой_таблицы>
...;
```

Табличный оператор `UNPIVOT`, как и его противоположность — `PIVOT`, работает в контексте инструкции `FROM`. На вход ему подается таблица или табличное выражение (в нашем случае `EmpCustOrders`). Внутри круглых скобок определяются атрибуты, которые будут хранить содержимое и имена исходных столбцов (`qty` и `custid`), а также список этих имен (**A**, **B**, **C** и **D**). После скобок необходимо указать псевдоним, который будет назначен таблице, полученной в результате работы этого оператора.

Ниже представлен готовый запрос, который отменяет разворачивание данных с помощью встроенных средств языка T-SQL.

```
SELECT empid, custid, qty
FROM dbo.EmpCustOrders
    UNPIVOT(qty FOR custid IN(A, B, C, D)) AS U;
```

Стоит отметить, что оператор `UNPIVOT` реализует те же логические этапы обработки, что были описаны ранее, — генерирование копий, извлечение элементов и удаление пересечений с отметками `NULL`. В отличие от стандартного решения последний этап здесь является обязательным.

Также нужно обратить внимание на то, что в результате отмены разворачивания таблица не возвращается к своему первоначальному виду. Это, скорее, преобразование развернутых значений в новый формат. Но если после отмены данной операции выполнить ее еще раз, получится та самая развернутая таблица. Другими словами, при первом разворачивании агрегирование приводит к частичной потере информации. После этого дальнейшие операции отмены и восстановления являются полностью обратимыми.

Теперь можете очистить БД, выполнив следующий код.

```
IF OBJECT_ID('dbo.EmpCustOrders', 'U') IS NOT NULL DROP TABLE dbo.
EmpCustOrders;
```

## Группирующие наборы

В текущем разделе рассматриваются группирующие наборы как таковые и их реализация в SQL Server.

**Группирующий набор** — это просто список атрибутов, по которым производится группирование. В языке SQL каждый агрегатный запрос традиционно может определять только один группирующий набор. Эта особенность проиллюстрирована в примерах, представленных ниже.

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid, custid;
```

```
SELECT empid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid;
```

```
SELECT custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY custid;
```

```
SELECT SUM(qty) AS sumqty
FROM dbo.Orders;
```

В первом примере группирующим набором являются столбцы **empid**, **custid**; во втором это столбец **empid**, в третьем — **custid**, а в четвертом мы имеем определение пустого набора — (). Этот код дает четыре результата — по одному для каждого запроса.

Теперь представьте, что вместо четырех отдельных наборов вы хотите получить один, в котором собраны все агрегированные данные. Для этого можно воспользоваться операцией **UNION ALL**, которая объединит все четыре результата. Так как все результирующие наборы должны иметь совместимую структуру с одинаковым количеством атрибутов, вам следует откорректировать имеющиеся запросы, добавив на место отсутствующих столбцов заглушки (например, отметки **NULL**). Вот как после этого будет выглядеть наш код.

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid, custid
```

```
UNION ALL
```

```
SELECT empid, NULL, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY empid
```

```
UNION ALL
```

```
SELECT NULL, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY custid
```

```
UNION ALL
```

```
SELECT NULL, NULL, SUM(qty) AS sumqty
FROM dbo.Orders;
```

В итоге мы получим единый результирующий набор с объединенными агрегатами для всех четырех запросов.

empid	custid	sumqty
2	A	52
3	A	20
1	B	20
2	B	27
1	C	34
3	C	22
3	D	30
1	NULL	54
2	NULL	79
3	NULL	72
NULL	A	72
NULL	B	47
NULL	C	56
NULL	D	30
NULL	NULL	205

(строк обработано: 15)

И хотя мы получили то, что хотели, у этого решения есть два существенных недостатка: большой объем кода и низкая производительность. Инструкция `GROUP BY` указывается для каждого группирующего набора, и если таких наборов будет много, выполнение запроса может существенно замедлиться. Кроме того, при обработке каждого запроса SQL Server должен будет заново сканировать исходную таблицу, что является пустой тратой ресурсов.

SQL Server поддерживает несколько стандартных инструментов, которые позволяют определять множественные группирующие наборы в одном и том же запросе. Речь идет об инструкциях `GROUPING SETS`, `CUBE` и `ROLLUP`, которые используются на этапе `GROUP BY`, а также о функциях `GROUPING` и `GROUPING_ID`.

## Вложенная инструкция `GROUPING SETS`

`GROUPING SETS` — это мощное дополнение к инструкции `GROUP BY`, которое в основном используется для вывода и хранения данных. С его помощью можно создавать множественные группирующие наборы в рамках одного запроса. Для этого их нужно перечислить внутри скобок через запятую. Каждый набор состоит из списка атрибутов, которые также разделены запятыми и заключены в скобки. Запрос, представленный ниже, создает четыре группирующих набора: `(empid, custid)`, `(empid)`, `(custid)` и `()`.

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY
    GROUPING SETS
    (
        (empid, custid),
        (empid),
        (custid),
        ()
    );
```



Этот пример делает то же самое, что и предыдущий; он объединяет результирующие наборы четырех агрегатных запросов и возвращает точно такой же результат. Однако у него есть два важных преимущества. Во-первых, как видно невооруженным взглядом, он намного более компактный. Во-вторых, SQL Server не должен будет сканировать исходную таблицу при формировании каждого группирующего набора.

## Вложенная инструкция CUBE

Вложенная инструкция CUBE используется на этапе GROUP BY и дает возможность определять множественные группирующие наборы в сокращенном виде. Внутри скобок этой инструкции через запятую должны быть перечислены атрибуты, все возможные комбинации которых будут возвращаться в виде результата. Например, CUBE(a, b, c) — то же самое, что GROUPING SETS((a, b, c), (a, b), (a, c), (b, c), (a), (b), (c), () ). Множество всех подмножеств, которые могут быть созданы из конкретного множества элементов, называют **булеаном**. Иначе говоря, инструкция CUBE генерирует булеан группирующих наборов на основе заданного множества атрибутов.

В предыдущем примере с помощью инструкции GROUPING SETS мы определили четыре группирующих набора: (empid, custid), (empid), (custid) и (). Вместо этого можно было использовать выражение CUBE(empid, custid). При этом запрос выглядел бы следующим образом.

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

## Вложенная инструкция ROLLUP

Вложенная инструкция ROLLUP во многом похожа на CUBE. Однако она генерирует не все группирующие наборы, которые можно получить на основе заданных атрибутов, а некое их подмножество. Инструкция ROLLUP исходит из того, что входящие атрибуты имеют иерархию. Если выражение CUBE(a, b, c) генерирует восемь группирующих наборов, то инструкция ROLLUP с теми же параметрами вернет только четыре; учитывая иерархию a, b, c, она вернет то же самое, что и GROUPING SETS((a, b, c), (a, b), (a), () ).

Представьте, например, что вам необходимо получить общее количество товара для всех группирующих наборов, которые можно определить на основе иерархии «год заказа → месяц заказа → день заказа». Для этого проще всего воспользоваться инструкцией GROUPING SETS, перечислив четыре возможных варианта.

```
GROUPING SETS(
    (YEAR(orderdate), MONTH(orderdate), DAY(orderdate)),
    (YEAR(orderdate), MONTH(orderdate)),
    (YEAR(orderdate)),
    () )
```

Но если применить инструкцию ROLLUP, код получится куда более компактным.

```
ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate))
```

Ниже представлен полноценный запрос.

```
SELECT
    YEAR(orderdate) AS orderyear,
    MONTH(orderdate) AS ordermonth,
    DAY(orderdate) AS orderday,
    SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY ROLLUP(YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

Запустив его, вы получите следующий результат.

orderyear	ordermonth	orderday	sumqty
-----	-----	-----	-----
2007	4	18	22
2007	4	NULL	22
2007	8	2	10
2007	8	NULL	10
2007	12	24	32
2007	12	NULL	32
2007	NULL	NULL	64
2008	1	9	40
2008	1	18	14
2008	1	NULL	54
2008	2	12	12
2008	2	NULL	12
2008	NULL	NULL	66
2009	2	12	10
2009	2	16	20
2009	2	NULL	30
2009	4	18	15
2009	4	NULL	15
2009	9	7	30
2009	9	NULL	30
2009	NULL	NULL	75
NULL	NULL	NULL	205

## Функции GROUPING и GROUPING\_ID

Если при выполнении одного запроса вы получаете несколько группирующих наборов, у вас должна быть возможность связывать их с итоговыми строками, то есть определять для каждой строки связанный с ней набор. В этом нет ничего сложного, если все группирующие элементы имеют ограничение NOT NULL. Рассмотрим, например, следующий запрос.

```
SELECT empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

Результат, который он возвращает, показан ниже.

empid	custid	sumqty
-----	-----	-----
2	A	52
3	A	20

NULL	A	72
1	B	20
2	B	27
NULL	B	47
1	C	34
3	C	22
NULL	C	56
3	D	30
NULL	D	30
NULL	NULL	205
1	NULL	54
2	NULL	79
3	NULL	72

Поскольку столбцы `empid` и `custid` в таблице `dbo.Orders` имеют ограничение `NOT NULL`, отметки `NULL` в них могут выступать только в качестве заглушек, сигнализируя о том, что они не входили в текущий группирующий набор. Поэтому все строки, у которых столбцы `empid` и `custid` равны `NULL`, связываются с набором (`empid`, `custid`). Если отметка `NULL` хранится только в столбце `custid`, строка попадает в набор (`empid`). Некоторые люди предпочитают использовать другие отметки (например, `ALL`), чтобы указать, что столбцы не допускают неопределенных значений. Это облегчает создание отчетов.

Если же в таблице нет ограничения `NOT NULL`, отметка `NULL` может быть как содержимым столбца, так и заглушкой для элементов, которые не входят в группирующий набор. Чтобы точно определить, с чем мы имеем дело, можно воспользоваться функцией `GROUPING`. Она принимает название столбца и возвращает одно из двух значений: 0 — если значение входит в группирующий набор, и 1 — если не входит.



**ПРИМЕЧАНИЕ**

Тот факт, что функция `GROUPING` возвращает 1, когда элемент не является частью группирующего набора, кажется мне странным. Я считаю, что в данном случае было бы логичнее возвращать 0, а 1 (аналог значения `TRUE`) оставить для элементов, которые входят в набор. Но такова текущая реализация, и вы должны четко понимать, как она работает.

Приведем в качестве примера следующий запрос, который вызывает функцию `GROUPING` для каждого группирующего элемента.

```
SELECT
    GROUPING(empid) AS grpemp,
    GROUPING(custid) AS grpcust,
    empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

Результат его выполнения показан ниже.

grpemp	grpcust	empid	custid	sumqty
0	0	2	A	52
0	0	3	A	20
1	0	NULL	A	72
0	0	1	B	20
0	0	2	B	27

1	0	NULL	B	47
0	0	1	C	34
0	0	3	C	22
1	0	NULL	C	56
0	0	3	D	30
1	0	NULL	D	30
1	1	NULL	NULL	205
0	1	1	NULL	54
0	1	2	NULL	79
0	1	3	NULL	72

(строк обработано: 15)

Теперь вам не нужно использовать отметки NULL, чтобы определять связь между итоговыми строками и группирующими наборами. Например, мы точно можем сказать, что в набор (empid, custid) входят все строки, в которых для столбцов **grtemp** и **grpcust** возвращается значение 0. Если 0 возвращается только для столбца **grtemp**, строка входит в набор (empid).

SQL Server также поддерживает функцию под названием **GROUPING\_ID**, которая может еще больше упростить процесс связывания итоговых строк с группирующими наборами. В качестве входящих аргументов выступают все атрибуты, попавшие хотя бы в один набор, например **GROUPING\_ID(a, b, c, d)**; результатом будет целочисленная битовая карта, в которой каждый переданный элемент представлен отдельным битом, то есть крайнему справа элементу соответствует крайний справа бит. Например, битовая карта набора (a, b, c, d) равна 0 ( $0 \times 8 + 0 \times 4 + 0 \times 2 + 0 \times 1$ ). Группирующий набор (a, c) будет представлен числом 5 ( $0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$ ).

В отличие от **GROUPING** функция **GROUPING\_ID** может принять все группирующие элементы за один вызов, как показано ниже.

```
SELECT
    GROUPING_ID(empid, custid) AS groupingset,
    empid, custid, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY CUBE(empid, custid);
```

Результат выполнения этого запроса будет следующим.

groupingset	empid	custid	sumqty
0	2	A	52
0	3	A	20
2	NULL	A	72
0	1	B	20
0	2	B	27
2	NULL	B	47
0	1	C	34
0	3	C	22
2	NULL	C	56
0	3	D	30
2	NULL	D	30
3	NULL	NULL	205
1	1	NULL	54
1	2	NULL	79
1	3	NULL	72

Теперь вы с легкостью можете определить, с каким группирующим набором связана та или иная строка. Число 0 (00 в двоичном виде) соответствует набору (`empid`, `custid`); число 1 (01) представляет набор (`empid`); число 2 (10) указывает на (`custid`); число 3 (11) связано с пустым набором ().

## В заключение

В этой главе были рассмотрены оконные функции, разворачивание данных (и его отмена), а также возможности, связанные с группирующими наборами.

Оконные функции позволяют проводить вычисления на основе наборов данных, используя наиболее гибкие и эффективные методы. Они имеют множество сфер применения, поэтому вам определенно стоит потратить время на то, чтобы как следует их изучить.

Мы рассмотрели как стандартный, так и нестандартный подходы к разворачиванию данных (и его отмену). В нестандартном варианте используются операторы `PIVOT` и `UNPIVOT`, встроенные в язык T-SQL; их главным преимуществом перед штатными средствами языка SQL является более компактный код.

SQL Server поддерживает несколько важных инструментов для гибкой и эффективной работы с группирующими наборами: вложенные инструкции `GROUPING SETS`, `CUBE` и `ROLLUP`, а также функции `GROUPING` и `GROUPING_ID`.

## Упражнения

В этом разделе рассматриваются упражнения, которые помогут вам закрепить материал, пройденный в главе 7. Все примеры подразумевают обращение к таблице `dbo.Orders`, которую вы создали и заполнили с помощью кода, представленного в листинге 7.1.

### 1

Напишите запрос к таблице `dbo.Orders`, который вычисляет для каждого клиентского заказа значения функций `RANK` и `DENSE_RANK`, с секционированием по столбцу `custid` и сортировкой по `qty`.

- Используется таблица `dbo.Orders`.
- Ожидаемый результат:

<code>custid</code>	<code>orderid</code>	<code>qty</code>	<code>rnk</code>	<code>drnk</code>
A	30001	10	1	1
A	40005	10	1	1
A	10001	12	3	2
A	40001	40	4	3
B	20001	12	1	1

B	30003	15	2	2
B	10005	20	3	3
C	10006	14	1	1
C	20002	20	2	2
C	30004	22	3	3
D	30007	30	1	1

## 2

Напишите запрос к таблице `dbo.Orders`, который вычисляет для каждой строки разницу между количеством товара в текущем и предыдущем, а также в текущем и следующем заказах.

- Используется таблица `dbo.Orders`.
- Ожидаемый результат:

custid	orderid	qty	diffprev	diffnext
-----				
A	30001	10	NULL	-2
A	10001	12	2	-28
A	40001	40	28	30
A	40005	10	-30	NULL
B	10005	20	NULL	8
B	20001	12	-8	-3
B	30003	15	3	NULL
C	30004	22	NULL	8
C	10006	14	-8	-6
C	20002	20	6	NULL
D	30007	30	NULL	NULL

## 3

Напишите запрос к таблице `dbo.Orders`, возвращающий строку для каждого сотрудника, столбец для каждого года, в котором выполнялись заказы, и количество заказов в каждом году и по каждому сотруднику.

- Используется таблица `dbo.Orders`.
- Ожидаемый результат:

empid	cnt2007	cnt2008	cnt2009
-----			
1	1	1	1
2	1	2	1
3	2	0	2

## 4

Запустите следующий код, чтобы создать и заполнить таблицу `EmpYearOrders`.

```
USE TSQL2012;

IF OBJECT_ID('dbo.EmpYearOrders', 'U')
  IS NOT NULL DROP TABLE dbo.EmpYearOrders;
```

```
CREATE TABLE dbo.EmpYearOrders
(
    empid INT NOT NULL
        CONSTRAINT PK_EmpYearOrders PRIMARY KEY,
    cnt2007 INT NULL,
    cnt2008 INT NULL,
    cnt2009 INT NULL
);

INSERT INTO dbo.EmpYearOrders(empid, cnt2007, cnt2008, cnt2009)
SELECT empid, [2007] AS cnt2007, [2008] AS cnt2008, [2009] AS cnt2009
FROM (SELECT empid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
      PIVOT(COUNT(orderyear)
            FOR orderyear IN([2007], [2008], [2009])) AS P;

SELECT * FROM dbo.EmpYearOrders;
```

Результат выполнения этого запроса представлен ниже.

empid	cnt2007	cnt2008	cnt2009
1	1	1	1
2	1	2	1
3	2	0	2

Теперь напишите запрос к таблице **EmpYearOrders**, который отменяет разворачивание данных и возвращает строку с количеством заказов для каждого сотрудника и за каждый год. Исключите из результата строки, в которых количество заказов равно 0 (в данном случае это актуально для сотрудника под номером 3, поскольку он не обработал ни одного заказа в 2008 г.).

- Используется таблица **EmpYearOrders**
- Ожидаемый результат:

empid	orderyear	numorders
1	2007	1
1	2008	1
1	2009	1
2	2007	1
2	2008	2
2	2009	1
3	2007	2
3	2009	2

5

Напишите запрос к таблице **dbo.Orders**, который возвращает общее количество заказанного товара для каждого из следующих наборов: (employee, customer, and order year), (employee and order year) и (customer and order year). В результат должен попасть столбец, который однозначно идентифицирует группирующий набор, что связан с текущей строкой.

- Используется таблица `dbo.Orders`.
- Ожидаемый результат:

groupingset	empid	custid	orderyear	sumqty
0	2	A	2007	12
0	3	A	2007	10
4	NULL	A	2007	22
0	2	A	2008	40
4	NULL	A	2008	40
0	3	A	2009	10
4	NULL	A	2009	10
0	1	B	2007	20
4	NULL	B	2007	20
0	2	B	2008	12
4	NULL	B	2008	12
0	2	B	2009	15
4	NULL	B	2009	15
0	3	C	2007	22
4	NULL	C	2007	22
0	1	C	2008	14
4	NULL	C	2008	14
0	1	C	2009	20
4	NULL	C	2009	20
0	3	D	2009	30
4	NULL	D	2009	30
2	1	NULL	2007	20
2	2	NULL	2007	12
2	3	NULL	2007	32
2	1	NULL	2008	14
2	2	NULL	2008	52
2	1	NULL	2009	20
2	2	NULL	2009	15
2	3	NULL	2009	40

(строк обработано: 29)

Закончив с упражнениями, очистите БД с помощью следующего кода.

```
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
```

## Решения

Здесь приводятся решения для упражнений, представленных в главе 7.

### 1

Это упражнение довольно тривиальное. Все, что от вас требуется, — это знание синтаксиса оконных ранжирующих функций. Решение, представленное ниже, возвращает для каждого заказа значения функций `RANK` и `DENSE_RANK`, с секционированием по столбцу `custid` и сортировкой по `qty`.

```
SELECT custid,orderid,qty,
       RANK() OVER(PARTITION BY custid ORDER BY qty) AS rnk,
       DENSE_RANK() OVER(PARTITION BY custid ORDER BY qty) AS drnk
FROM   dbo.Orders;
```



## 2

Оконные функции со смещением, LAG и LEAD, позволяют получать элементы из предыдущей и следующей строк соответственно, учитывая заданное секционирование и порядок сортировки. В данном упражнении необходимо выполнить вычисления с каждым клиентским заказом, поэтому оконное секционирование должно быть основано на столбце `custid`. Упорядочивание следует проводить по столбцу `orderdate` с дополнительным условием `orderid`. Ниже представлено полноценное решение.

```
SELECT custid, orderid, qty,  
       qty - LAG(qty) OVER(PARTITION BY custid  
                           ORDER BY orderdate, orderid) AS diffprev,  
       qty - LEAD(qty) OVER(PARTITION BY custid  
                             ORDER BY orderdate, orderid) AS diffnext  
FROM dbo.Orders;
```

Это хороший пример того, как в одном выражении можно смешать оконные функции с отдельными элементами строки.

## 3

Для выполнения разворачивания данных первым делом необходимо определить элементы, которые задействуются в группировании и распределении, а также агрегатную функцию и аргумент для нее. Потом все это достаточно разместить в правильном порядке внутри «шаблонного» запроса — и не важно, используется при этом стандартное решение или встроенный оператор `PIVOT`.

В данном упражнении группирующим элементом является сотрудник (`empid`), распределение выполняется по году заказа (`YEAR(orderdate)`), а в качестве агрегата выступает функция `COUNT`; однако с агрегирующим элементом не все так просто. Нам нужно подсчитать подходящие строки и заказы — нас не интересуют конкретные атрибуты. Другими словами, мы можем указать любой столбец — главное, чтобы он не допускал хранения отметок `NULL`, потому что агрегатные функции игнорируют неопределенные значения (они могут привести к некорректным результатам).

Если выбор входящего атрибута для функции `COUNT` не имеет никакого значения, почему бы не указать те же атрибуты, что были заданы в распределяющем элементе? Так вы сможете использовать год заказа и для распределения, и для агрегации.

Теперь, когда мы определились с элементами для разворачивания, можно перейти к написанию итогового решения. Без использования оператора `PIVOT` оно будет выглядеть следующим образом.

```
USE TSQL2012;  
  
SELECT empid,  
       COUNT(CASE WHEN orderyear = 2007 THEN orderyear END) AS cnt2007,  
       COUNT(CASE WHEN orderyear = 2008 THEN orderyear END) AS cnt2008,  
       COUNT(CASE WHEN orderyear = 2009 THEN orderyear END) AS cnt2009  
FROM (SELECT empid, YEAR(orderdate) AS orderyear  
      FROM dbo.Orders) AS D  
GROUP BY empid;
```

Как вы помните, в выражении CASE по умолчанию используется инструкция ELSE NULL. Следовательно, мы будем получать определенные значения только в том случае, когда заказы отвечают поставленным требованиям, то есть если они обработаны текущим сотрудником в текущем году. И именно эти значения потом будут учитываться агрегатной функцией COUNT.

Стоит отметить, что в стандартном решении не обязательно использовать табличные выражения, но чтобы не делать вызов функции YEAR(orderdate) больше одного раза, я назначил ей псевдоним orderyear.

Ниже представлен вариант решения с использованием встроенного оператора PIVOT.

```
SELECT empid, [2007] AS cnt2007, [2008] AS cnt2008, [2009] AS cnt2009
FROM (SELECT empid, YEAR(orderdate) AS orderyear
      FROM dbo.Orders) AS D
      PIVOT(COUNT(orderyear)
            FOR orderyear IN([2007], [2008], [2009])) AS P;
```

Как видите, достаточно расставить элементы в правильном порядке.

Если вы хотите назначить целевым столбцам собственные имена, не полагаясь на содержимое таблицы, можете указать соответствующие псевдонимы в списке SELECT. В этом запросе я пометил столбцы [2007], [2008] и [2009] как cnt2007, cnt2008 и cnt2009.

## 4

В этом упражнении вам необходимо отменить разворачивание исходных столбцов cnt2007, cnt2008 и cnt2009. На выходе у вас должно быть два столбца — orderyear (для хранения годов, которые представлены именами исходных атрибутов) и numorders (для хранения исходных значений). За основу можно взять решение, которое демонстрировалось ранее в этой главе, но с небольшими изменениями.

В примерах, которые приводились в этой главе, отметки NULL использовались для обозначения лишних значений, а строки с ними отбрасывались на этапе фильтрации. Таблица EmpYearOrders не содержит отметок NULL, но в некоторых ее ячейках хранятся нули; при этом согласно условию мы должны отфильтровать строки с нулевым количеством заказов. В стандартном решении вместо выражения IS NOT NULL нужно использовать предикат numorders <> 0. Ниже представлен код, в котором используется инструкция VALUES.

```
SELECT *
FROM (SELECT empid, orderyear,
      CASE orderyear
        WHEN 2007 THEN cnt2007
        WHEN 2008 THEN cnt2008
        WHEN 2009 THEN cnt2009
      END AS numorders
      FROM dbo.EmpYearOrders
      CROSS JOIN (VALUES(2007),(2008),(2009)) AS Years (orderyear)) AS D
WHERE numorders <> 0;
```

Что касается встроенного оператора UNPIVOT, нужно отметить, что при его использовании автоматически удаляются значения NULL. Однако нули остаются на месте — их вы должны убрать самостоятельно с помощью инструкции WHERE (см. ниже).

```
SELECT empid, CAST(RIGHT(orderyear, 4) AS INT) AS orderyear, numorders
FROM dbo.EmpYearOrders
UNPIVOT(numorders FOR orderyear IN(cnt2007, cnt2008, cnt2009)) AS U
WHERE numorders <> 0;
```

Обратите внимание на выражение в списке SELECT, с помощью которого генерируется итоговый столбец **orderyear**: CAST(RIGHT(orderyear, 4) AS INT). Исходные атрибуты имеют имена cnt2007, cnt2008 и cnt2009. В результате применения оператора UNPIVOT они превращаются в значения 'cnt2007', 'cnt2008' и 'cnt2009', хранящиеся в столбце **orderyear**. Это выражение нужно для того, чтобы извлечь четыре последних символа, представляющих год заказа, и преобразовать их в целое число. В стандартном решении нам удалось обойтись без этих манипуляций, поскольку константы, которые использовались в табличном выражении Years, изначально были целочисленными.

## 5

Для тех, кто понимает принцип работы группирующих наборов, это упражнение должно показаться довольно тривиальным. Для создания списка группирующих наборов можно воспользоваться вложенной инструкцией GROUPING SETS, а чтобы сгенерировать для них уникальные идентификаторы и привязать их к каждой отдельной строке, подойдет функция GROUPING\_ID. Полноценный запрос представлен ниже.

```
SELECT
    GROUPING_ID(empid, custid, YEAR(Orderdate)) AS groupingset,
    empid, custid, YEAR(Orderdate) AS orderyear, SUM(qty) AS sumqty
FROM dbo.Orders
GROUP BY
    GROUPING SETS
    (
        (empid, custid, YEAR(orderdate)),
        (empid, YEAR(orderdate)),
        (custid, YEAR(orderdate))
    );
```

Указанные группирующие наборы не являются ни булеаном, ни свертком какого-то множества атрибутов. Следовательно, код, представленный выше, нельзя сделать еще более компактным, даже если воспользоваться вложенными инструкциями CUBE или ROLLUP.

## Глава 8

# ИЗМЕНЕНИЕ ДАННЫХ

SQL поддерживает набор инструкций, известных под аббревиатурой DML. Как нетрудно догадаться по названию, их главная функция заключается в изменении данных. Вопреки распространенному мнению, язык DML предназначен не только для обработки, но и для извлечения содержимого таблиц. В его состав входят такие инструкции, как SELECT, INSERT, UPDATE, DELETE, TRUNCATE и MERGE. До этого момента мы в основном рассматривали только инструкцию SELECT. В данной главе давайте сосредоточим внимание на операциях по обработке данных. Помимо стандартных средств мы также затронем некоторые особенности языка T-SQL.

Чтобы не вносить изменений в уже имеющиеся данные, большинство примеров в этой главе предполагает создание и заполнение таблиц в контексте схемы `dbo`.

## Добавление данных

Язык T-SQL предоставляет несколько команд для добавления данных в таблицы: INSERT VALUES, INSERT SELECT, INSERT EXEC, SELECT INTO и BULK INSERT. После того как мы их рассмотрим, вы сможете познакомиться с инструментами для автоматического создания ключей — объектами последовательности и свойством `identity`.

### Команда INSERT VALUES

Команда INSERT VALUES используется для добавления в таблицу строк, основанных на заданных значениях. Чтобы иметь возможность упражняться в применении этой и других инструкций, мы создадим и заполним таблицу `Orders` в схеме `dbo`. Для этого нужно запустить следующий код.

```
USE TSQL2012;

IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;

CREATE TABLE dbo.Orders
(
    orderid INT NOT NULL
        CONSTRAINT PK_Orders PRIMARY KEY,
    orderdate DATE NOT NULL
        CONSTRAINT DFT_orderdate DEFAULT(SYSDATETIME()),
    empid INT NOT NULL,
    custid VARCHAR(10) NOT NULL
)
```

На примере запроса, представленного ниже, демонстрируется использование команды `INSERT VALUES` для добавления в таблицу `Orders` одиночной строки.

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
VALUES(10001, '20090212', 3, 'A');
```

Перечисление имен целевых столбцов сразу после имени самой таблицы не является обязательным, однако такой подход позволяет контролировать связь между значениями и столбцами, не полагаясь на порядок их следования в определении таблицы (или в той команде, с помощью которой в последний раз изменялась ее структура).

Microsoft SQL Server будет использовать то значение столбца, которое было указано. Если вы этого не сделали, ядро БД попытается воспользоваться значением по умолчанию, которое находится в определении столбца. Если его там нет, ячейке будет присвоена отметка `NULL`. Если же столбец не допускает хранения этих отметок и не генерирует значения автоматически, ваша команда `INSERT` завершится неудачно. В качестве примера использования выражений или значений по умолчанию можно рассмотреть команду, представленную ниже. Она добавляет строку в таблицу `Orders`, не указывая явно значения для столбца `orderdate`; при этом в определении `orderdate` содержится выражение по умолчанию (`SYSDATETIME`).

```
INSERT INTO dbo.Orders(orderid, empid, custid)
VALUES(10002, 5, 'B');
```

SQL Server версий 2008 и 2012 поддерживает улучшенную инструкцию `VALUES`, которая позволяет указывать через запятую сразу несколько строк. Например, следующая команда добавляет четыре строки в таблицу `Orders`.

```
INSERT INTO dbo.Orders
(orderid, orderdate, empid, custid)
VALUES
(10003, '20090213', 4, 'B'),
(10004, '20090214', 1, 'A'),
(10005, '20090213', 1, 'C'),
(10006, '20090215', 3, 'C');
```

Этот код обрабатывается как атомарная операция, то есть если хотя бы одна строка не сможет попасть в таблицу, все остальные тоже не будут добавлены.

У инструкции `VALUES` есть еще одна интересная особенность. Она может выступать в качестве стандартного конструктора табличных значений при создании производных таблиц. Ниже показан пример.

```
SELECT *
FROM ( VALUES
      (10003, '20090213', 4, 'B'),
      (10004, '20090214', 1, 'A'),
      (10005, '20090213', 1, 'C'),
      (10006, '20090215', 3, 'C') )
AS O(orderid, orderdate, empid, custid);
```

После конструктора табличных значений идут псевдоним таблицы (в нашем случае `O`) и псевдонимы целевых столбцов, перечисленные внутри скобок. Этот запрос генерирует следующий результат.

orderid	orderdate	empid	custid
10003	20090213	4	B
10004	20090214	1	A
10005	20090213	1	C
10006	20090215	3	C

## Команда INSERT SELECT

Команда `INSERT SELECT` добавляет в таблицу набор строк, полученных в результате выборки. Синтаксис почти такой же, как у команды `INSERT VALUES`, только с запросом `SELECT` вместо инструкции `VALUES`. Например, следующий код добавляет в таблицу `dbo.Orders` результат выполнения запроса к таблице `Sales.Orders`, который возвращает заказы, доставленные в Великобританию.

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
  SELECT orderid, orderdate, empid, custid
  FROM Sales.Orders
  WHERE shipcountry = 'Великобритания';
```

Команда `INSERT SELECT` также позволяет указывать имена целевых столбцов (те рекомендации относительно именования, которые я давал ранее, остаются в силе). Вы должны предоставить значения для всех столбцов, которые не заполняются автоматически; принцип использования значений по умолчанию и отметок `NULL` остается таким же, как в случае с `INSERT VALUES`. Команда `INSERT SELECT` выполняется как атомарная операция, поэтому если хотя бы одна строка не сможет попасть в таблицу, все остальные строки тоже не будут добавлены.

До того как в `SQL Server` появилась улучшенная инструкция `VALUES`, для создания виртуальных таблиц на основе значений приходилось прибегать к многократному вызову инструкции `SELECT`, каждый из которых возвращал одну строку, и затем объединять результат с помощью операции `UNION ALL`. Точно такой же подход можно использовать и в контексте команды `INSERT SELECT`. Например, следующий запрос добавляет в таблицу `Orders` четыре строки, основанные на заранее заданных значениях.

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
  SELECT 10007, '20090215', 2, 'B' UNION ALL
  SELECT 10008, '20090215', 1, 'C' UNION ALL
  SELECT 10009, '20090216', 2, 'C' UNION ALL
  SELECT 10010, '20090216', 3, 'A';
```

Однако этот синтаксис считается нестандартным, потому что он подразумевает использование команд `SELECT` без инструкций `FROM`. В спецификации языка `SQL` предусмотрен более предпочтительный вариант — применение конструктора табличных значений, основанного на инструкции `VALUES`.

## Команда INSERT EXEC

Команда `INSERT EXEC` предназначена для добавления в таблицу строк, возвращаемых хранимыми процедурами или динамическими пакетами (информацию

о которых вы сможете найти в главе 10). По принципу своей работы и синтаксису команда `INSERT EXEC` очень похожа на `INSERT SELECT`, только вместо инструкции `SELECT` в ней используется `EXEC`.

Следующий код создает хранимую процедуру под названием `Sales.usp_getorders`, которая возвращает заказы, отправленные в определенную страну (указанную с помощью параметра `@country`).

```
IF OBJECT_ID('Sales.usp_getorders', 'P') IS NOT NULL
    DROP PROC Sales.usp_getorders;
GO

CREATE PROC Sales.usp_getorders
    @country AS NVARCHAR(40)
AS

SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE shipcountry = @country;
GO
```

Чтобы увидеть, как работает эта хранимая процедура, передайте ей параметр «Франция».

```
EXEC Sales.usp_getorders @country = 'Франция';
```

Вы получите следующий результат.

orderid	orderdate	empid	custid
10248	2006-07-04 00:00:00.000	5	85
10251	2006-07-08 00:00:00.000	3	84
10265	2006-07-25 00:00:00.000	2	7
10274	2006-08-06 00:00:00.000	6	85
10295	2006-09-02 00:00:00.000	2	85
10297	2006-09-04 00:00:00.000	5	7
10311	2006-09-20 00:00:00.000	1	18
10331	2006-10-16 00:00:00.000	9	9
10334	2006-10-21 00:00:00.000	8	84
10340	2006-10-29 00:00:00.000	1	9
...			

(строк обработано: 77)

Команда `INSERT EXEC` позволяет направить результат, полученный из хранимой процедуры, прямо в таблицу `dbo.Orders`.

```
INSERT INTO dbo.Orders(orderid, orderdate, empid, custid)
EXEC Sales.usp_getorders @country = 'Франция';
```

## Команда **SELECT INTO**

Команда `SELECT INTO` создает целевую таблицу и заполняет ее результирующим набором, который возвращается запросом. Она не входит в стандарты ISO и ANSI, но поддерживается языком T-SQL. Ее нельзя использовать для добавле-

ния данных в уже имеющиеся таблицы. Она имеет вид `INTO <имя_целевой_таблицы>` и должна быть указана непосредственно перед инструкцией `FOR` того запроса, который генерирует результирующий набор. Например, следующий код создает таблицу под названием `dbo.Orders` и копирует в нее все содержимое таблицы `Sales.Orders`.

```
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;

SELECT orderid, orderdate, empid, custid
INTO dbo.Orders
FROM Sales.Orders;
```

Структура и содержимое целевой таблицы основываются на источнике результирующего набора. Это касается имен и типов столбцов, поддержки отметок `NULL` и свойства `identity`. Существует четыре аспекта исходной таблицы, которые игнорируются командой `SELECT INTO`: ограничения, индексы, триггеры и разрешения.



#### ПРИМЕЧАНИЕ

На момент написания этой книги сервис Windows Azure SQL Database не поддерживал кучи (таблицы без кластеризованных индексов). В то же время команда `SELECT INTO` создает именно кучу, поскольку она не копирует индексы (в том числе кластеризованные). В связи с этим данная команда не поддерживается в SQL Database. Чтобы получить тот же результат, необходимо использовать связку команд `CREATE TABLE` и `INSERT SELECT`.

Одним из преимуществ команды `SELECT INTO` является то, что она выполняется в режиме минимального журналирования (если только для свойства `Recovery Model` не определено значение `FULL`). Этот режим, по сравнению с обычным, обеспечивает крайне высокую скорость работы. То же самое касается команды `INSERT SELECT`, но у нее есть дополнительные условия. Подробности можно найти в электронном справочнике в разделе «Предварительные условия для минимального ведения журнала массового импорта данных» по адресу [msdn.microsoft.com/ru-ru/library/ms190422.aspx](https://msdn.microsoft.com/ru-ru/library/ms190422.aspx).

Если вы хотите использовать команду `SELECT INTO` в сочетании с операциями для работы с наборами, инструкция `INTO` должна находиться непосредственно перед этапом `FROM`. Например, следующий код создает таблицу под названием `Locations` и копирует в нее результаты выполнения операции `EXCEPT`; в итоге мы должны получить места проживания клиентов, но не сотрудников.

```
IF OBJECT_ID('dbo.Locations', 'U') IS NOT NULL DROP TABLE dbo.Locations;

SELECT country, region, city
INTO dbo.Locations
FROM Sales.Customers

EXCEPT

SELECT country, region, city
FROM HR.Employees;
```



## Команда BULK INSERT

Команда `BULK INSERT` добавляет в таблицу данные, взятые из файла. При этом необходимо указать целевую таблицу, исходный файл и параметры, включая тип данных (например, `char` или `native`), признак конца поля и другие (их полный список можно найти в документации).

Следующий код добавляет все содержимое файла `c:\temp\orders.txt` в таблицу `dbo.Orders`; в качестве типа данных выступает `char`, признаком конца поля служит запятая, а для обозначения конца записи используется символ перевода строки (`'\n'`).

```
BULK INSERT dbo.Orders FROM 'c:\temp\orders.txt'
WITH
(
    DATAFILETYPE = 'char',
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '\n'
);
```

Стоит отметить, что для выполнения этого кода вам необходимо поместить файл `orders.txt` (он предоставляется вместе с исходным кодом к данной книге) в директорию `c:\temp`.

В некоторых ситуациях команда `BULK INSERT` может работать в режиме минимального журналирования, но для этого необходимо выполнить определенные условия. Подробно о данном вопросе можно прочитать в электронном справочнике в разделе «Предварительные условия для минимального ведения журнала массового импорта данных».

## Свойство identity и объект последовательности

В SQL Server реализовано два встроенных средства для генерирования ключей: свойство столбца `identity` и объект последовательности. Свойство `identity` поддерживается в SQL Server с незапамятных времен. Несмотря на неплохую работу в некоторых ситуациях, у него есть множество недостатков. Объект последовательности, реализованный в SQL Server 2012, решает большинство из этих проблем.

### Свойство identity

SQL Server позволяет определить для столбца любого числового типа свойство `identity` (допускаются нулевые или положительные значения без дробной части). Это свойство автоматически генерирует значения во время выполнения команды `INSERT`, основываясь на аргументах `seed` (точка отсчета) и `increment` (длина шага), которые должны быть указаны в определении столбца. Обычно этот инструмент используется для создания **суррогатных ключей**, которые генерируются на системном уровне и не зависят от программных данных.

Возьмем для примера следующий код, который создает таблицу `dbo.T1`.

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;

CREATE TABLE dbo.T1
(
    keycol INT NOT NULL IDENTITY(1, 1)
```

```

        CONSTRAINT PK_T1 PRIMARY KEY,
        datacol VARCHAR(10) NOT NULL
        CONSTRAINT CHK_T1_datacol CHECK(datacol LIKE '[A-Za-z]%' )
    );

```

Таблица содержит столбец под названием **keycol**, для которого было указано свойство **identity** (начальное значение и длина шага равны 1). В таблице также имеется столбец для хранения строк — **datacol**; из-за ограничения **CHECK** он может принимать только те значения, которые начинаются с алфавитного символа.

При использовании команды **INSERT** столбец со свойством **identity** нужно игнорировать, как будто его вовсе не существует. Например, следующий код успешно добавляет в таблицу три строки, хотя значения указываются только для столбца **datacol**.

```

INSERT INTO dbo.T1(datacol) VALUES('AAAAA');
INSERT INTO dbo.T1(datacol) VALUES('CCCCC');
INSERT INTO dbo.T1(datacol) VALUES('BBBBB');

```

SQL Server автоматически сгенерировал значения для столбца **keycol**. Чтобы в этом убедиться, выполните запрос к таблице.

```
SELECT * FROM dbo.T1;
```

Вы должны получить следующий результат.

keycol	datacol
1	AAAAA
2	CCCCC
3	BBBBB

Во время запроса к таблице у вас есть возможность обращаться к столбцу со свойством **identity** по его имени (в нашем случае **keycol**). Но это также легко сделать, используя универсальный идентификатор **\$identity**.

Данный подход поможет отобразить из таблицы **T1** значения столбца со свойством **identity**, как показано ниже.

```
SELECT $identity FROM dbo.T1;
```

Результат будет следующим.

keycol
1
2
3

При добавлении в таблицу новой строки SQL Server генерирует ключ, исходя из текущего значения **identity** и заданного инкремента. Если вам нужно получить только что сгенерированное значение (например, чтобы добавить дочерние строки в ссылающуюся таблицу), вы можете воспользоваться функциями **@@identity** или **SCOPE\_IDENTITY**. Функция **@@identity** существует уже довольно давно;

она возвращает последний ключ, сгенерированный во время текущей сессии, независимо от контекста (например, текущая процедура и триггер, вызванный командой INSERT, — это разные контексты). Функция `SCOPE_IDENTITY` делает то же самое, но только с учетом текущего контекста (например, для одной и той же процедуры).

Следующий код добавляет строку в таблицу `T1`, получает только что сгенерированный ключ с помощью функции `SCOPE_IDENTITY` и выводит его в виде переменной.

```
DECLARE @new_key AS INT;

INSERT INTO dbo.T1(datacol) VALUES('AAAAA');

SET @new_key = SCOPE_IDENTITY();

SELECT @new_key AS new_key
```

Если вы выполнили все примеры кода, представленные в этом разделе, результат будет следующим.

```
new_key
-----
4
```

Как уже говорилось выше, функции `@@identity` и `SCOPE_IDENTITY` возвращают последний ключ, который был сгенерирован в рамках текущей сессии. То есть добавление строки во время параллельной сессии никак не повлияет на результат. Но если вы хотите знать последнее значение `identity`, которое было сохранено в таблице, вам нужно использовать функцию `IDENT_CURRENT`. Создайте новую сессию и запустите следующий код.

```
SELECT
    SCOPE_IDENTITY() AS [SCOPE_IDENTITY],
    @@identity AS [@@identity],
    IDENT_CURRENT('dbo.T1') AS [IDENT_CURRENT];
```

Результат представлен ниже.

SCOPE_IDENTITY	@@identity	IDENT_CURRENT
-----	-----	-----
NULL	NULL	4

Обе функции, `@@identity` и `SCOPE_IDENTITY`, вернули отметки `NULL`, потому что в текущей сессии не было создано ни одного ключа. Функция `IDENT_CURRENT` вернула 4 — последнее значение `identity`, которое было сохранено в таблице, вне зависимости от сессии.

В оставшейся части этого раздела мы обсудим некоторые важные аспекты свойства `identity`.

Изменения текущего значения `identity` не отменяются в случае неудачного выполнения транзакции. К примеру, запустите следующую команду INSERT, которая конфликтует с ограничением `CHECK`, объявленным в таблице.

```
INSERT INTO dbo.T1(datacol) VALUES('12345');
```

Добавление строки завершится ошибкой.

Сообщение 547, уровень 16, состояние 0, строка 1

Конфликт инструкции INSERT с ограничением CHECK "CHK\_T1\_datacol". Конфликт произошел в базе данных "TSQL2012", таблица "dbo.T1", column 'datacol'.

Выполнение данной инструкции было прервано.

Тем не менее значение `identity` было инкрементировано, и теперь оно равно 5 — эти изменения не были автоматически отменены из-за ошибки. Следовательно, следующая операция добавления сгенерирует значение 6.

```
INSERT INTO dbo.T1(datacol) VALUES('EEEEEE');
```

Выполните запрос к таблице.

```
SELECT * FROM dbo.T1;
```

Как видите, у нас получился промежуток между значениями 4 и 6.

keycol	datacol
1	AAAAA
2	CCCCC
3	BBBBB
4	AAAAA
6	EEEEEE

Если вас не устраивает наличие таких промежутков, вам стоит подумать об использовании альтернативного механизма.

Еще один важный аспект свойства `identity` заключается в том, что вы не можете добавлять его в уже имеющиеся столбцы или удалять. Его использование допускается только в рамках команд `CREATE TABLE` и `ALTER TABLE` (в случае создания нового столбца). Однако SQL Server все же позволяет устанавливать собственные значения для столбцов со свойством `identity`; для этого команду `INSERT` необходимо выполнять в рамках сессии, созданной с использованием параметра `IDENTITY_INSERT`. Тем не менее возможность изменения ранее сохраненных значений полностью исключена.

Чтобы увидеть, как это работает, выполните следующий код, который вставляет в таблицу `T1` строку, явно присваивая столбцу `keycol` значение 5.

```
SET IDENTITY_INSERT dbo.T1 ON;  
INSERT INTO dbo.T1(keycol, datacol) VALUES(5, 'FFFFF');  
SET IDENTITY_INSERT dbo.T1 OFF;
```

Интересно, что текущее значение `identity` меняется только в том случае, когда число, переданное в запросе, больше того, которое уже находится в таблице. Перед тем как мы запустили вышеприведенный код, текущее значение `identity` было равно 6, а в команде `INSERT` было указано число 5, поэтому никаких изменений не произошло. Таким образом, запустив функцию `IDENT_CURRENT`, вы получите 6, а не 5. Код, представленный ниже, вернет 7.

```
INSERT INTO dbo.T1(datacol) VALUES('GGGGG');
```

Теперь извлеките содержимое таблицы T1.

```
SELECT * FROM dbo.T1;
```

Вы получите следующий результат.

keycol	datacol
1	AAAAA
2	CCCCC
3	BBBBB
4	AAAAA
5	FFFFF
6	EEEEEE
7	GGGGG

Важно понимать, что свойство `identity` само по себе не обеспечивает уникальность значений столбца. Вы можете задавать собственные значения, если установите параметр `IDENTITY_INSERT` в режим `ON`, и при этом не будет выполняться никакой проверки на наличие дубликатов. Можно также повторно установить текущее значение `identity`, воспользовавшись командой `DBCC CHECKIDENT` (описание ее синтаксиса находится в соответствующем разделе электронного справочника). Если вам необходимо гарантировать уникальность столбца со свойством `identity`, объявите для него первичный ключ или ограничение `UNIQUE`.

## Объект последовательности

Реализация объектов последовательности появилась в SQL Server 2012. Это альтернативный способ создания ключей. Данный механизм считается стандартным и поддерживается в некоторых других СУРБД, что упрощает процесс миграции. Объект последовательности обладает значительно большей гибкостью, нежели свойство `identity`, благодаря чему его использование во многих ситуациях является предпочтительным.

Одно из преимуществ объектов последовательности заключается в том, что в отличие от свойства `identity` они не привязаны к конкретному столбцу таблицы и представляют собой автономные сущности БД. Объекты передаются в функцию, результаты выполнения которой можно использовать по своему усмотрению. Это означает, что ключи, полученные из одного объекта, не будут конфликтовать даже в контексте нескольких таблиц.

Для создания объектов последовательности используется команда `CREATE SEQUENCE`. Имя последовательности является единственным обязательным параметром, но имейте в виду, что значения, предлагаемые по умолчанию, могут вас не устроить. В качестве стандартного типа используется `BIGINT`; его можно заменить на любой числовой тип (со значениями большими или равными нулю), прибегнув к выражению `AS <тип>`. Например, если вы хотите получить целочисленную последовательность, укажите `AS INT`.

В отличие от свойства `identity` объекты последовательности позволяют задавать минимальное (`MINVALUE <val>`) и максимальное (`MAXVALUE <val>`) значения. По умолчанию используется тот диапазон, который поддерживается типом объекта. Например, для типа `INT` это будут значения от `-2,147,483,648` до `2,147,483,647`.

Кроме того, в отличие от свойства `identity` объекты последовательности поддерживают циклическое повторение — специально для этого предусмотрен параметр `CYCLE`.

У объектов последовательности и свойства `identity` есть кое-что общее — это поддержка начального значения (`START WITH <val>`) и величины шага (`INCREMENT BY <val>`); по умолчанию используются соответственно минимальное значение (`MINVALUE`) и число 1.

Представьте, к примеру, что вам нужно сгенерировать последовательность для дальнейшего создания идентификаторов. Вы хотите, чтобы она была целочисленной, начиналась с единицы, имела максимальное значение, допустимое в заданном типе, и поддерживала циклическое повторение. Величина шага и минимальное значение должны быть равны 1.

Команда `CREATE SEQUENCE`, с помощью которой проще всего реализовать такую последовательность, представлена ниже.

```
CREATE SEQUENCE dbo.SeqOrderIDs AS INT
    MINVALUE 1
    CYCLE;
```

Тип, минимальное значение и поддержку циклического повторения необходимо указывать вручную. Остальные параметры имеют значения по умолчанию, которые нам подходят.

Объекты последовательности также поддерживают параметр кеширования (`CACHE <val> | NO CACHE`), который сигнализирует о том, сколько значений должно записываться на диск. Чем меньший показатель вы выберете, тем быстрее будут генерироваться ваши ключи (в среднем), но тем больше данных потеряете в случае непредвиденного сбоя в SQL Server. Значение по умолчанию не задокументировано и, как следствие, может меняться от версии к версии.

Используя команду `ALTER SEQUENCE`, можно изменять любые другие параметры последовательности (`MINVAL <val>`, `MAXVAL <val>`, `RESTART WITH <val>`, `INCREMENT BY <val>`, `CYCLE | NO CYCLE` или `CACHE <val> | NO CACHE`). Представьте, к примеру, что вам нужно отменить циклическое повторение для таблицы `dbo.SeqOrderIDs`. Для этого вы можете изменить определение последовательности, используя следующую команду.

```
ALTER SEQUENCE dbo.SeqOrderIDs
    NO CYCLE;
```

Чтобы сгенерировать новое значение в рамках определенной последовательности, вызовите функцию `NEXT VALUE FOR <имя_последовательности>`. Это действительно функция, как бы странно она ни выглядела. Ее можно разместить внутри инструкции `SELECT`, как показано ниже.

```
SELECT NEXT VALUE FOR dbo.SeqOrderIDs;
```

Этот код генерирует следующий результат.

```
-----
1
```

Стоит отметить, что для создания нового значения объекты последовательности, в отличие от свойства `identity`, не требуют добавления строки в таблицу. Вы можете сохранить результат выполнения функции в переменной и затем использовать его по своему усмотрению. Чтобы это проиллюстрировать, создадим таблицу `T1`, воспользовавшись следующим кодом.

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;

CREATE TABLE dbo.T1
(
    keycol INT NOT NULL
        CONSTRAINT PK_T1 PRIMARY KEY,
    datacol VARCHAR(10) NOT NULL
);
```

Код, представленный ниже, генерирует в рамках нашей последовательности новое значение и сохраняет его в переменной, которая затем используется в команде `INSERT` для добавления строки в таблицу.

```
DECLARE @neworderid AS INT = NEXT VALUE FOR dbo.SeqOrderIDs;
INSERT INTO dbo.T1(keycol, datacol) VALUES(@neworderid, 'a');

SELECT * FROM dbo.T1;
```

Результат будет следующим.

keycol	datacol
2	a

Тот же подход вы можете применять и при использовании новых ключей в связанных строках, которые должны добавляться в другую таблицу.

Если вам не нужно сохранять вновь сгенерированный элемент последовательности для каких-то дальнейших действий, вы можете передать функцию `NEXT VALUE FOR` непосредственно в команду `INSERT`, как показано ниже.

```
INSERT INTO dbo.T1(keycol, datacol)
VALUES(NEXT VALUE FOR dbo.SeqOrderIDs, 'b');

SELECT * FROM dbo.T1;
```

Этот код возвращает следующий результат.

keycol	datacol
2	a
3	b

В отличие от свойства `identity` объекты последовательности позволяют генерировать значения в рамках команды `UPDATE`.

```
UPDATE dbo.T1
SET keycol = NEXT VALUE FOR dbo.SeqOrderIDs;

SELECT * FROM dbo.T1;
```

Вот каким должен быть результат.

keycol	datacol
4	a
5	b

Для получения информации о последовательности необходимо выполнить запрос к представлению `sys.sequences`. Например, чтобы найти текущее значение последовательности `SeqOrderIDs`, выполните следующий код.

```
SELECT current_value
FROM sys.sequences
WHERE OBJECT_ID = OBJECT_ID('dbo.SeqOrderIDs');
```

Результат показан ниже.

current_value
5

Поддержка последовательностей в SQL Server выходит за рамки стандарта и является более развитой, чем в других СУРБД. Одна из дополнительных возможностей позволяет управлять порядком следования присвоенных значений во время добавления нескольких строк; для этого предусмотрена инструкция `OVER`, которая во многом напоминает уже знакомые нам оконные функции. Ниже представлен пример.

```
INSERT INTO dbo.T1(keycol, datacol)
SELECT
    NEXT VALUE FOR dbo.SeqOrderIDs OVER(ORDER BY hiredate),
    LEFT(firstname, 1) + LEFT(lastname, 1)
FROM HR.Employees;

SELECT * FROM dbo.T1;
```

Этот код генерирует следующий результат.

keycol	datacol
4	a
5	b
6	ДЛ
7	СД
8	ДФ
9	ИП
10	СБ
11	ПС
12	ПК
13	МК
14	ЗД

Кроме того, язык T-SQL позволяет использовать функцию `NEXT VALUE FOR` в качестве значения по умолчанию. Ниже показан пример.

```
ALTER TABLE dbo.T1
ADD CONSTRAINT DFT_T1_keycol
```



```
DEFAULT (NEXT VALUE FOR dbo.SeqOrderIDs)
FOR keycol;
```

Теперь при добавлении строк в таблицу вам не нужно указывать значение для столбца **keycol**.

```
INSERT INTO dbo.T1(datacol) VALUES('c');
```

```
SELECT * FROM dbo.T1;
```

Этот код возвращает следующий результат.

keycol	datacol
4	a
5	b
6	ДЛ
7	СД
8	ДФ
9	ИП
10	СВ
11	ПС
12	РК
13	МК
14	ЗД
15	С

Это существенное преимущество по сравнению со свойством `identity` — вы можете добавлять и удалять ограничение `DEFAULT` из уже существующих таблиц.

Еще одна дополнительная возможность языка T-SQL позволяет выделять в рамках последовательности целый диапазон значений, используя хранимую процедуру под названием `sp_sequence_get_range`. Представьте, что вам необходимо получить какой-то определенный отрезок последовательности; самым простым способом будет добавить к имеющимся значениям размер заданного диапазона. Вы вызываете процедуру, указываете нужный вам размер и затем извлекаете из результата первое значение в диапазоне (а также сопутствующую информацию). В примере, представленном ниже, мы вызываем процедуру и запрашиваем диапазон для 1000 значений последовательности.

```
DECLARE @first AS SQL_VARIANT;

EXEC sys.sp_sequence_get_range
    @sequence_name = N'dbo.SeqOrderIDs',
    @range_size = 1000,
    @range_first_value = @first OUTPUT ;

SELECT @first;
```

Если запустить этот код дважды, обнаружится, что разница между первыми значениями в двух вызовах составляет 1000.

По аналогии со свойством `identity` объекты последовательности не исключают наличия промежутков между элементами. Если проводимая транзакция по какой-то причине завершится ошибкой, значение последовательности не будет приведено к исходному виду.

Закончив с примерами, очистите БД с помощью следующего кода.

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
IF OBJECT_ID('dbo.SeqOrderIDs', 'So') IS NOT NULL DROP SEQUENCE dbo.
SeqOrderIDs;
```

## Удаление данных

Язык T-SQL поддерживает две команды для удаления строк из таблицы — DELETE и TRUNCATE. О них и пойдет речь в этом разделе. Запросы, которые приводятся здесь в качестве примеров, направлены к копиям таблиц Customers и Orders. Чтобы создать эти копии в рамках схемы dbo, выполните следующий код.

```
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
IF OBJECT_ID('dbo.Customers', 'U') IS NOT NULL DROP TABLE dbo.Customers;
```

```
CREATE TABLE dbo.Customers
(
    custid          INT NOT NULL,
    companyname     NVARCHAR(40) NOT NULL,
    contactname     NVARCHAR(30) NOT NULL,
    contacttitle    NVARCHAR(30) NOT NULL,
    address         NVARCHAR(60) NOT NULL,
    city           NVARCHAR(15) NOT NULL,
    region         NVARCHAR(15) NULL,
    postalcode     NVARCHAR(10) NULL,
    country        NVARCHAR(15) NOT NULL,
    phone         NVARCHAR(24) NOT NULL,
    fax           NVARCHAR(24) NULL,
    CONSTRAINT PK_Customers PRIMARY KEY(custid)
);

CREATE TABLE dbo.Orders
(
    orderid        INT          NOT NULL,
    custid         INT          NULL,
    empid         INT          NOT NULL,
    orderdate      DATETIME     NOT NULL,
    requireddate   DATETIME     NOT NULL,
    shippeddate     DATETIME     NULL,
    shipperid      INT          NOT NULL,
    freight        MONEY        NOT NULL,
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname       NVARCHAR(40) NOT NULL,
    shipaddress    NVARCHAR(60) NOT NULL,
    shipcity       NVARCHAR(15) NOT NULL,
    shipregion     NVARCHAR(15) NULL,
    shippostalcode NVARCHAR(10) NULL,
    shipcountry    NVARCHAR(15) NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(orderid),
    CONSTRAINT FK_Orders_Customers FOREIGN KEY(custid)
        REFERENCES dbo.Customers(custid)
);
GO
```

```
INSERT INTO dbo.Customers SELECT * FROM Sales.Customers;
INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
```

## Команда DELETE

DELETE — это стандартная команда, которая используется для удаления данных из таблицы с учетом предиката. Спецификацией языка SQL предусмотрено только две инструкции, которые можно использовать в сочетании с этой командой, — FROM (для задания имени целевой таблицы) и WHERE (для определения условия). Удаляются лишь те строки, для которых указанный предикат возвращает TRUE.

Например, следующий код удаляет из таблицы `dbo.Orders` все заказы, которые были размещены до 2007 г.

```
DELETE FROM dbo.Orders
WHERE orderdate < '20070101';
```

В результате выполнения этого запроса SQL Server сообщит об удалении 152 строк.

```
(строк обработано: 152)
```

Нужно отметить, что данное сообщение появится только в том случае, если параметр сессии NOCOUNT установлен в режим OFF (это его состояние по умолчанию). Если же данный параметр включен (ON), SQL Server Management Studio всего лишь отrapортирует об успешном выполнении запроса.

Все действия команды DELETE полностью записываются в журнал. Таким образом, не стоит ждать от нее высокой производительности при удалении большого количества записей.

## Команда TRUNCATE

Команда TRUNCATE удаляет из таблицы все ее содержимое. В отличие от DELETE она не поддерживает никаких фильтров. Например, следующий код удаляет все строки из таблицы `dbo.T1`.

```
TRUNCATE TABLE dbo.T1;
```

Главное преимущество команды TRUNCATE заключается в том, что она выполняется в режиме минимального журналирования, что обеспечивает значительный прирост в производительности. Например, удаление содержимого таблицы с миллионом строк с ее помощью займет секунды, тогда как у команды DELETE на то же действие уйдут минуты или даже часы. Вопреки распространенному заблуждению, команда TRUNCATE является полностью транзакционной и в случае выполнения операции ROLLBACK ее действие может быть отменено.

Кроме того, команда TRUNCATE, в отличие от DELETE, сбрасывает содержимое свойства `identity` к его исходному состоянию.

Команду TRUNCATE нельзя применять к таблицам, на которые ссылается внешний ключ, даже если ссылающаяся таблица пуста, а сам ключ не активирован. Единственный выход — удалить все внешние ключи, которые указывают на нужную вам таблицу.

Иногда случаются курьезные ситуации, когда таблица удаляется по ошибке. Представьте, что у вас открыты соединения как с рабочей, так и с отладочной БД и вы случайно отправили свой код не туда. Команды TRUNCATE и DROP выполняются настолько быстро, что прежде чем вы осознаете ошибку, транзакция уже будет произведена. Чтобы избежать подобных неприятностей, создайте внешний ключ, который будет ссылаться на вашу рабочую таблицу, защищая ее от удаления. Вы можете даже его деактивировать — это ничего не изменит. Как я уже упоминал выше, важен сам факт наличия внешнего ключа.

## Команда DELETE в сочетании с оператором JOIN

Язык T-SQL поддерживает нестандартный синтаксис команды DELETE, основанный на применении оператора JOIN. Само соединение служит в качестве фильтра (за счет предиката ON). С его помощью также легко получить доступ к атрибутам соответствующих строк из другой таблицы, используя инструкцию WHERE. Благодаря этому вы можете удалять строки, исходя из содержимого атрибутов, взятых из другой таблицы.

Например, следующий код удаляет заказы, размещенные американскими клиентами.

```
DELETE FROM O
FROM dbo.Orders AS O
      JOIN dbo.Customers AS C
      ON O.custid = C.custid
WHERE C.country = N'США';
```

Здесь, как и в случае с командой SELECT, в первую очередь выполняется инструкция FROM (та, что указана во второй строчке). После этого последовательно обрабатываются инструкции WHERE и DELETE. Данный код можно «прочитать» или интерпретировать так: «Запрос соединяет таблицы Orders (псевдоним O) и Customers (псевдоним C), основываясь на соответствии клиентских идентификаторов, которые они хранят. Затем запрос отбирает только те заказы, которые были размещены клиентами из США. В конце запрос удаляет подходящие строки из таблицы O (то есть из Orders)».

Наличие сразу двух инструкций FROM выглядит довольно странно. Чтобы окончательно не запутаться, можете считать, что вы выполняете команду SELECT с соединением. Начните с инструкции FROM, в которой проводится соединение, затем перейдите к инструкции WHERE и в самом конце замените SELECT на DELETE, указывая псевдоним той части соединения, которая подлежит удалению.

Как уже упоминалось выше, команда DELETE, основанная на соединении, является нестандартной. Если вы хотите придерживаться спецификации языка SQL, вместо соединений лучше использовать вложенные запросы. Например, код, представленный ниже, выполняет ту же задачу.

```
DELETE FROM dbo.Orders
WHERE EXISTS
  (SELECT *
   FROM dbo.Customers AS C
   WHERE Orders.Custid = C.Custid
   AND C.Country = 'США');
```

Эта команда удаляет все заказы (таблица **Orders**), которые были сделаны американскими клиентами (таблица **Customers**).

Оба запроса, скорее всего, будут обрабатываться совершенно одинаково; следовательно, никакой разницы в производительности быть не должно. Так зачем же использовать нестандартный синтаксис? Это дело вкуса: одним нравятся соединения, а другие предпочитают вложенные запросы. Я рекомендую придерживаться спецификации языка SQL и отклоняться от нее только в случае возникновения реальной необходимости, например если это дает существенный прирост производительности.

Закончив с примерами, очистите БД с помощью следующего кода.

```
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
IF OBJECT_ID('dbo.Customers', 'U') IS NOT NULL DROP TABLE dbo.Customers;
```

## Обновление данных

В этом разделе мы рассмотрим использование команды UPDATE, которая позволяет обновлять строки, хранящиеся в таблице. Язык T-SQL поддерживает две реализации данной команды, одна из которых является нестандартной (на основе соединений).

Запросы, которые приводятся здесь в качестве примеров, направлены к копиям таблиц **Orders** и **OrderDetails**. Чтобы создать эти копии в рамках схемы **dbo**, выполните следующий код.

```
IF OBJECT_ID('dbo.OrderDetails', 'U') IS NOT NULL DROP TABLE dbo.
OrderDetails;
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
```

```
CREATE TABLE dbo.Orders
(
    orderid          INT          NOT NULL,
    custid           INT          NULL,
    empid            INT          NOT NULL,
    orderdate        DATETIME     NOT NULL,
    requireddate     DATETIME     NOT NULL,
    shippeddate       DATETIME     NULL,
    shipperid        INT          NOT NULL,
    freight          MONEY        NOT NULL,
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname         NVARCHAR(40) NOT NULL,
    shipaddress      NVARCHAR(60) NOT NULL,
    shipcity         NVARCHAR(15) NOT NULL,
    shipregion       NVARCHAR(15) NULL,
    shippostalcode  NVARCHAR(10) NULL,
    shipcountry      NVARCHAR(15) NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);
```

```
CREATE TABLE dbo.OrderDetails
(
    orderid  INT          NOT NULL,
```

```

productid INT          NOT NULL,
unitprice MONEY        NOT NULL
    CONSTRAINT DFT_OrderDetails_unitprice DEFAULT(0),
qty SMALLINT          NOT NULL
    CONSTRAINT DFT_OrderDetails_qty DEFAULT(1),
discount NUMERIC(4, 3) NOT NULL
    CONSTRAINT DFT_OrderDetails_discount DEFAULT(0),
CONSTRAINT PK_OrderDetails PRIMARY KEY(orderid, productid),
CONSTRAINT FK_OrderDetails_Orders FOREIGN KEY(orderid)
    REFERENCES dbo.Orders(orderid),
CONSTRAINT CHK_discount CHECK (discount BETWEEN 0 AND 1),
CONSTRAINT CHK_qty CHECK (qty > 0),
CONSTRAINT CHK_unitprice CHECK (unitprice >= 0)
);
GO

INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
INSERT INTO dbo.OrderDetails SELECT * FROM Sales.OrderDetails;

```

## Команда UPDATE

Стандартный синтаксис команды UPDATE позволяет обновлять некое подмножество строк таблицы. Для определения этого подмножества используется предикат внутри инструкции WHERE. Значения или выражения, которые присваиваются отдельным столбцам, разделяются запятыми и указываются с помощью инструкции SET.

Например, следующий код увеличивает скидку для продукта под номером 51 на 5 %.

```

UPDATE dbo.OrderDetails
    SET discount = discount + 0.05
WHERE productid = 51;

```

Чтобы увидеть изменения, вы можете запустить команду SELECT с тем же фильтром до и после обновления. Позже в этой главе я продемонстрирую еще один подход к отслеживанию изменений, который подразумевает использование команды UPDATE совместно с инструкцией OUTPUT.

SQL Server 2008 и SQL Server 2012 поддерживают составные операторы присваивания: += (с суммированием), -= (с вычитанием), \*= (с умножением), /= (с делением), %= (присваивание по модулю). Они позволяют сокращать выражения, в которых выполняется присваивание значений. Например, в предыдущем запросе содержится код discount = discount + 0.05; вместо него можно использовать выражение discount += 0.05, как показано ниже.

```

UPDATE dbo.OrderDetails
    SET discount += 0.05
WHERE productid = 51;

```

При выполнении команды UPDATE необходимо помнить о таком важном аспекте языка SQL, как одновременные операции. Мы говорили об этом еще в главе 2, но тот же принцип применим и к команде UPDATE. Как вы помните, все выражения, которые находятся на одном логическом этапе обработки,

выполняются одновременно. Чтобы понять значимость этого факта, рассмотрим следующий запрос.

```
UPDATE dbo.T1
SET col1 = col1 + 10, col2 = col1 + 10;
```

Предположим, что перед обновлением некая строка содержит в столбцах `col1` и `col2` значения 100 и 200 соответственно. Можете ли вы сказать, чему будут равны эти значения после запуска вышеприведенного кода?

Если не брать в расчет одновременность выполнения операций и принять, что обработка производится слева направо, то можно предположить, что столбец `col1` будет равен 110, а `col2` — 120. На самом деле обе операции присваивания происходят в один и тот же момент времени; это означает, что в обоих случаях будет использовано исходное значение `col1`. Следовательно, оба столбца будут равны 110.

Теперь, учитывая тот факт, что операции выполняются одновременно, попробуйте написать команду `UPDATE`, которая меняет местами значения `col1` и `col2`. В большинстве языков программирования, в которых выражения обрабатываются в каком-то определенном порядке (обычно слева направо), для этого, как правило, используется временная переменная. Однако в языке SQL решение выглядит еще проще.

```
UPDATE dbo.T1
SET col1 = col2, col2 = col1;
```

В обеих операциях присваивания используются исходные значения столбцов, поэтому временная переменная здесь не нужна.

## Команда `UPDATE` в сочетании с оператором `JOIN`

Язык T-SQL поддерживает нестандартный синтаксис команды `UPDATE`, основанный на соединении. Как и в случае с операцией удаления, соединение играет роль фильтра.

Синтаксис практически ничем не отличается от обычного соединения, включая инструкции `FROM` и `WHERE`; разница только в том, что вместо `SELECT` необходимо указать команду `UPDATE`. После ключевого слова `UPDATE` должно следовать имя таблицы, которую нужно обновить (за одну команду можно обновить только одну таблицу), а также инструкция `SET` с операциями присваивания.

Команда `UPDATE`, приведенная в листинге 8.1, увеличивает скидку на 5 % для всех заказов, размещенных клиентом под номером 1.

### Листинг 8.1. Команда `UPDATE`, основанная на соединении

```
UPDATE OD
SET discount += 0.05
FROM dbo.OrderDetails AS OD
JOIN dbo.Orders AS O
ON OD.orderid = O.orderid
WHERE O.custid = 1;
```

Интерпретация этого запроса начинается с инструкции FROM, затем идет инструкция WHERE, и только в самом конце выполняется команда UPDATE. При этом происходит соединение таблиц OrderDetails (псевдоним OD) и Orders (псевдоним O) с учетом соответствия идентификаторов заказов, которые они хранят. Затем запрос отбирает только те строки, в которых идентификатор клиента равен 1. В конце команда UPDATE обращается к таблице OD (OrderDetails) и увеличивает скидку на 5 %.

Чтобы достичь того же результата с помощью стандартного кода, нужно заменить соединение вложенным запросом, как показано ниже.

```
UPDATE dbo.OrderDetails
    SET discount += 0.05
WHERE EXISTS
    (SELECT * FROM dbo.Orders AS O
     WHERE O.orderid = OrderDetails.orderid
     AND O.custid = 1);
```

Инструкция WHERE, размещенная в этом коде, отбирает информацию только о тех заказах, которые были размещены клиентом под номером 1. В этом конкретном случае оба примера, скорее всего, будут интерпретированы одинаково, без ощутимой разницы в производительности. Опять же выбор зависит исключительно от того, что вам нравится больше — соединения или вложенные запросы. Но, как я уже упоминал раньше при рассмотрении команды DELETE, стандартному синтаксису следует отдавать предпочтение, если только у вас нет весомой причины этого не делать. В данном случае такой причины нет.

Однако бывают ситуации, когда соединения обрабатываются быстрее, чем вложенные запросы. Они не только выполняют фильтрацию, но и дают доступ к атрибутам из других таблиц, которые можно использовать для присваивания значений в инструкции SET. В принципе, вложенные запросы тоже позволяют это делать, но тогда каждое обращение к другой таблице будет выполняться отдельно — по крайней мере, так работает современное ядро SQL Server.

Давайте рассмотрим нестандартный вариант команды UPDATE на примере следующего кода.

```
UPDATE T1
    SET col1 = T2.col1,
        col2 = T2.col2,
        col3 = T2.col3
FROM dbo.T1 JOIN dbo.T2
    ON T2.keycol = T1.keycol
WHERE T2.col4 = 'ABC';
```

Эта команда соединяет таблицы T1 и T2, основываясь на соответствии столбцов T1.keycol и T2.keycol. Инструкция WHERE отбирает только те строки, в которых столбец T2.col4 содержит значение 'ABC'. В качестве объекта для обновления выступает таблица T1, поскольку она указана в команде UPDATE; инструкция SET присваивает ее столбцам, col1, col2 и col3, значения, которые хранятся в соответствующих столбцах таблицы T2.

Попытка решить эту задачу стандартными средствами с использованием вложенных запросов приводит к увеличению объема кода.



```
UPDATE dbo.T1
  SET col1 = (SELECT col1
              FROM dbo.T2
              WHERE T2.keycol = T1.keycol),

  col2 = (SELECT col2
          FROM dbo.T2
          WHERE T2.keycol = T1.keycol),

  col3 = (SELECT col3
          FROM dbo.T2
          WHERE T2.keycol = T1.keycol)
WHERE EXISTS
  (SELECT *
   FROM dbo.T2
   WHERE T2.keycol = T1.keycol
        AND T2.col4 = 'ABC');
```

Мало того, что этот вариант получился запутанным (в отличие от предыдущего), так еще и каждый вложенный запрос иницирует отдельное обращение к таблице T2. В связи с этим наблюдается падение производительности.

Стандартная версия языка SQL поддерживает конструкторы табличных значений (известные также как векторные выражения), которые были частично реализованы только в SQL Server 2012. Многие аспекты этих конструкторов до сих пор остаются недоступными, например их все еще нельзя использовать внутри инструкции SET при выполнении команды UPDATE, как показано ниже.

```
UPDATE dbo.T1

  SET (col1, col2, col3) =

    (SELECT col1, col2, col3
     FROM dbo.T2
     WHERE T2.keycol = T1.keycol)

WHERE EXISTS
  (SELECT *
   FROM dbo.T2
   WHERE T2.keycol = T1.keycol
        AND T2.col4 = 'ABC');
```

Но, как вы можете видеть, эта версия все равно выглядит более сложной, чем исходный вариант с соединением, потому что для фильтрации и извлечения атрибутов из другой таблицы требуется выполнение отдельных вложенных запросов.

## Обновление с присваиванием

Язык T-SQL поддерживает нестандартную версию команды UPDATE, которая вместе с обновлением данных в таблице присваивает значения переменным. Это позволяет избежать использования дополнительной инструкции SELECT.

Такой вариант команды UPDATE часто применяется при создании последовательностей и выполнении автоматической нумерации, когда столбцы со свойством `identity` и объекты последовательности попросту не подходят. Представьте, к примеру, что вам нужен механизм, который генерирует последовательные значе-

ния без промежутков. Для этого последнее использованное значение сохраняется в таблице, инкрементируется при обновлении и присваивается переменной.

Для начала запустите код, представленный ниже, чтобы создать таблицу **Sequence** со столбцом **val** и добавить в нее одну-единственную строку со значением 0 (первое значение последовательности будет равно 1).

```
IF OBJECT_ID('dbo.Sequences', 'U') IS NOT NULL DROP TABLE dbo.Sequences;

CREATE TABLE dbo.Sequences
(
    id VARCHAR(10) NOT NULL
        CONSTRAINT PK_Sequences PRIMARY KEY(id),
    val INT NOT NULL
);
INSERT INTO dbo.Sequences VALUES('SEQ1', 0);
```

Теперь, если вам нужно получить новое значение последовательности, вы можете запустить следующий код.

```
DECLARE @nextval AS INT;

UPDATE dbo.Sequences
    SET @nextval = val += 1
WHERE id = 'SEQ1';

SELECT @nextval;
```

Здесь мы объявляем переменную под названием **@nextval**. Затем с помощью специальной версии команды **UPDATE** увеличиваем значение столбца на 1, присваиваем его переменной и возвращаем в качестве результата. Операции присваивания внутри инструкции **SET** выполняются справа налево, то есть сначала столбцу **val** присваивается **val + 1**, после полученное значение сохраняется в переменной **@nextval**.

Такая версия команды **UPDATE** выполняется как атомарная операция и демонстрирует более высокую производительность по сравнению со связкой **UPDATE** и **SELECT**. Это связано с тем, что обращение к данным происходит всего один раз.

Закончив с примерами, очистите БД с помощью следующего кода.

```
IF OBJECT_ID('dbo.Sequences', 'U') IS NOT NULL DROP TABLE dbo.Sequences;
```

## Слияние данных

В SQL Server 2008 и SQL Server 2012 есть поддержка команды **MERGE**, которая позволяет изменять данные, выполняя с ними различные действия (**INSERT**, **UPDATE** и **DELETE**) на основе условной логики. Эта команда является стандартной, хотя в языке T-SQL у нее есть несколько дополнительных возможностей.

Запрос, состоящий из одной команды **MERGE**, обычно транслируется в комбинацию нескольких команд языка DML (**INSERT**, **UPDATE** и **DELETE**); при этом ключевое слово **MERGE** убирается. Такая операция позволяет уменьшить объем кода и улучшить производительность за счет уменьшения количества обращений к таблицам.

Чтобы продемонстрировать возможности команды `MERGE`, я воспользуюсь таблицами `dbo.Customers` и `dbo.CustomersStage`, создать и заполнить которые можно посредством кода, представленного в листинге 8.2.

**Листинг 8.2.** Код для создания и заполнения таблиц `dbo.Customers` и `dbo.CustomersStage`

```
IF OBJECT_ID('dbo.Customers', 'U') IS NOT NULL DROP TABLE dbo.Customers;
GO

CREATE TABLE dbo.Customers
(
    custid          INT          NOT NULL,
    companyname     VARCHAR(25) NOT NULL,
    phone           VARCHAR(20) NOT NULL,
    address         VARCHAR(50) NOT NULL,
    CONSTRAINT PK_Customers PRIMARY KEY(custid)
);

INSERT INTO dbo.Customers(custid, companyname, phone, address)
VALUES
    (1, 'cust 1', '(111) 111-1111', 'address 1'),
    (2, 'cust 2', '(222) 222-2222', 'address 2'),
    (3, 'cust 3', '(333) 333-3333', 'address 3'),
    (4, 'cust 4', '(444) 444-4444', 'address 4'),
    (5, 'cust 5', '(555) 555-5555', 'address 5');

IF OBJECT_ID('dbo.CustomersStage', 'U') IS NOT NULL DROP TABLE dbo.
CustomersStage;
GO

CREATE TABLE dbo.CustomersStage
(
    custid          INT          NOT NULL,
    companyname     VARCHAR(25) NOT NULL,
    phone           VARCHAR(20) NOT NULL,
    address         VARCHAR(50) NOT NULL,
    CONSTRAINT PK_CustomersStage PRIMARY KEY(custid)
);

INSERT INTO dbo.CustomersStage(custid, companyname, phone, address)
VALUES
    (2, 'AAAAA', '(222) 222-2222', 'address 2'),
    (3, 'cust 3', '(333) 333-3333', 'address 3'),
    (5, 'BBBBB', 'CCCCC', 'DDDDD'),
    (6, 'cust 6 (new)', '(666) 666-6666', 'address 6'),
    (7, 'cust 7 (new)', '(777) 777-7777', 'address 7');
```

Запустите следующий запрос, чтобы вывести содержимое таблицы `Customers`.

```
SELECT * FROM dbo.Customers;
```

Результат показан ниже.

custid	companyname	phone	address
1	cust 1	(111) 111-1111	address 1
2	cust 2	(222) 222-2222	address 2

3	cust 3	(333) 333-3333	address 3
4	cust 4	(444) 444-4444	address 4
5	cust 5	(555) 555-5555	address 5

Теперь выведите все строки таблицы **CustomersStage**.

```
SELECT * FROM dbo.CustomersStage;
```

Этот вопрос должен вернуть следующий результат.

custid	companyname	phone	address
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

В первом примере работы команды **MERGE**, который я скоро продемонстрирую, содержимое таблицы **CustomersStage** (источник) объединяется с таблицей **Customers** (цель). Если быть более точным, этот код будет добавлять записи о клиентах, которые еще не были сохранены, и обновлять атрибуты уже имеющихся записей.

Если вы уверенно владеете материалом, посвященным удалению и обновлению данных на основе соединений, то у вас не должно возникнуть проблем и с командой **MERGE**, которая имеет схожую семантику. Целевая таблица указывается после ключевого слова **MERGE**, а исходная — внутри инструкции **USING**. Как и в случае с оператором **JOIN**, условие объединения описывается с помощью предиката **ON**; оно определяет, какие строки исходной таблицы имеют соответствия в целевой, а какие нет. Действия, которые выполняются в случае наличия или отсутствия совпадения, должны быть указаны посредством инструкций **WHEN MATCHED THEN** и **WHEN NOT MATCHED THEN** соответственно.

Ниже представлен первый пример использования команды **MERGE**: добавление несуществующих и обновление уже имеющихся записей о клиентах.

```
MERGE INTO dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
ON TGT.custid = SRC.custid
WHEN MATCHED THEN
    UPDATE SET
        TGT.companyname = SRC.companyname,
        TGT.phone = SRC.phone,
        TGT.address = SRC.address
WHEN NOT MATCHED THEN
    INSERT (custid, companyname, phone, address)
    VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address);
```



#### ПРИМЕЧАНИЕ

Команда **MERGE** всегда должна заканчиваться точкой с запятой, хотя для большинства других команд в языке T-SQL это требование не является обязательным. Но если вы уже следуете рекомендациям по поводу разделения предложений, которые я давал ранее, вас это обстоятельство беспокоить не должно.

Этот код определяет таблицы **Customers** и **CustomersStage** в качестве цели и источника соответственно (во втором случае используется инструкция **USING**). Обратите внимание, что источнику и цели можно назначать псевдонимы (в нашем случае это **TGT** и **SRC**). С помощью предиката **TGT.custid = SRC.custid** определяются условия, при которых совпадение либо происходит, либо нет. В нашем случае совпадением считается ситуация, когда идентификатор клиента существует как в исходной, так и в целевой таблице. В противном случае совпадение отсутствует.

При наличии совпадения выполняется команда **UPDATE**, которая берет целевую строку и обновляет ее атрибуты **companyname**, **phone** и **address** исходными значениями. Здесь используется точно такой же синтаксис, как и при обычном обновлении, только имя целевой таблицы уже определено в рамках команды **MERGE**.

Если совпадение не обнаружено, выполняется команда **INSERT**, которая копирует строку из исходной таблицы в целевую. Ситуация с синтаксисом точно такая же, как в случае с командой **UPDATE**.

Запустив вышеприведенный пример, вы получите сообщение об изменении пяти строк.

(строк обработано: 5)

Речь идет о трех обновленных строках (клиенты с номерами 2, 3 и 5) и еще о двух, которые были добавлены (клиенты с номерами 6 и 7). Выполните запрос к таблице **Customers**, чтобы извлечь ее содержимое.

```
SELECT * FROM dbo.Customers;
```

Результат будет следующим.

custid	companyname	phone	address
1	cust 1	(111) 111-1111	address 1
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
4	cust 4	(444) 444-4444	address 4
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

Действия, которые выполняются при совпадении или несовпадении исходных и целевых строк, определяются инструкциями **WHEN MATCHED** и **WHEN NOT MATCHED** соответственно. В языке **T-SQL** есть еще одна инструкция, которая предусмотрена на тот случай, когда целевая строка не имеет соответствия в исходной таблице; она называется **WHEN NOT MATCHED BY SOURCE**. Представьте, что вам нужно изменить команду **MERGE** из предыдущего примера так, чтобы она удаляла строки из таблицы **Customers**, у которых нет соответствия в таблице **CustomersStage**. Для этого достаточно добавить инструкцию **WHEN NOT MATCHED BY SOURCE** с действием **DELETE**, как показано ниже.

```
MERGE dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
ON TGT.custid = SRC.custid
WHEN MATCHED THEN
```

```

UPDATE SET
    TGT.companyname = SRC.companyname,
    TGT.phone = SRC.phone,
    TGT.address = SRC.address
WHEN NOT MATCHED THEN
    INSERT (custid, companyname, phone, address)
    VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;

```

Выполните еще один запрос к таблице **Customers**, чтобы увидеть результат предыдущего объединения.

```
SELECT * FROM dbo.Customers;
```

Полученный результирующий набор представлен ниже; как видите, клиенты с номерами 1 и 4 были удалены.

custid	companyname	phone	address
2	AAAAA	(222) 222-2222	address 2
3	cust 3	(333) 333-3333	address 3
5	BBBBB	CCCCC	DDDDD
6	cust 6 (new)	(666) 666-6666	address 6
7	cust 7 (new)	(777) 777-7777	address 7

Если вернуться к первому примеру, в котором обновляются имеющиеся и добавляются новые записи о клиентах, можно заметить, что он написан не самым эффективным образом. Прежде чем перезаписывать атрибуты существующих строк, команда **MERGE** не проверяет, изменились ли они на самом деле. Это означает, что запись будет выполнена даже в том случае, если исходная и целевая строки являются идентичными. Возникшую проблему легко решить с помощью оператора **AND**, который позволяет добавить новый предикат; соответствующее действие будет выполнено только тогда, когда результатом проверки условия станет значение **TRUE**. Сам предикат нужно поместить внутрь инструкции **WHEN MATCHED AND**, которая проверяет, изменился ли хотя бы один атрибут (то есть стоит ли выполнять обновление). Весь код целиком будет выглядеть так.

```

MERGE dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
    ON TGT.custid = SRC.custid
WHEN MATCHED AND
    (TGT.companyname <> SRC.companyname
    OR TGT.phone <> SRC.phone
    OR TGT.address <> SRC.address) THEN
    UPDATE SET
        TGT.companyname = SRC.companyname,
        TGT.phone = SRC.phone,
        TGT.address = SRC.address
WHEN NOT MATCHED THEN
    INSERT (custid, companyname, phone, address)
    VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address);

```

Как видите, благодаря своим широким возможностям, команда **MERGE** позволяет не только повысить эффективность редактирования содержимого таблиц, но и уменьшить общий объем кода.

## Изменение данных с помощью табличных выражений

Действия, которые можно выполнять с табличными выражениями (производными таблицами, ОТВ, представлениями, пользовательскими встроенными ФТЗ), не ограничиваются только командой `SELECT`; SQL Server поддерживает и другие элементы языка DML (`INSERT`, `UPDATE`, `DELETE` и `MERGE`). Но если подумать, табличные выражения ничего не содержат — они являются всего лишь отражением исходных данных, которые хранятся в таблицах. Учитывая этот факт, редактирование табличных выражений можно считать изменением содержимого исходных таблиц с неким промежуточным этапом. Как и в случае с командой `SELECT`, использование других элементов языка DML приводит к разворачиванию табличного выражения, в результате чего взаимодействие выполняется непосредственно с атрибутами таблицы.

Редактирование данных через табличные выражения имеет несколько логических ограничений.

- Если запрос, определяющий табличное выражение, соединяет таблицы, вы можете влиять только на одну из сторон соединения, но не на обе сразу.
- Вы не можете обновить столбец, который является результатом вычислений; SQL Server не выполняет автоматическое извлечение исходных значений.
- Команда `INSERT` должна заполнять все столбцы исходной таблицы, значения для которых не генерируются автоматически. Столбец можно проигнорировать, если он допускает хранение отметок `NULL`, поддерживает значение по умолчанию, обладает свойством `identity` или имеет тип `ROWVERSION`.

Остальные требования можно найти в электронном справочнике. Как видите, они не лишены смысла.

Теперь, когда вы знаете, что данные можно редактировать через табличные выражения, встает закономерный вопрос: зачем это нужно? Одной из причин является упрощение отладки и поиска ошибок. Например, в листинге 8.1 содержится команда `UPDATE`.

```
UPDATE OD
  SET discount += 0.05
FROM dbo.OrderDetails AS OD
  JOIN dbo.Orders AS O
    ON OD.orderid = O.orderid
WHERE O.custid = 1;
```

Представьте, что вам нужно проанализировать полученный код на наличие каких-либо проблем. Сначала необходимо понять, какие строки будут изменены в результате его выполнения (при этом сами изменения вносить нельзя). Для этого вам следует заменить команду `UPDATE` на `SELECT` и после проведения анализа вернуть все обратно. Но вместо подобных сложных рокировок можно просто заменить целевую таблицу табличным выражением, основанным на команде `SELECT` и выполняющим операцию соединения. Ниже представлен пример с использованием ОТВ.

```

WITH C AS
(
    SELECT custid, OD.orderid,
           productid, discount, discount + 0.05 AS newdiscount
    FROM dbo.OrderDetails AS OD
         JOIN dbo.Orders AS O
           ON OD.orderid = O.orderid
    WHERE O.custid = 1
)
UPDATE C
    SET discount = newdiscount;

```

А вот вариант с производной таблицей.

```

UPDATE D
    SET discount = newdiscount
FROM ( SELECT custid, OD.orderid,
           productid, discount, discount + 0.05 AS newdiscount
    FROM dbo.OrderDetails AS OD
         JOIN dbo.Orders AS O
           ON OD.orderid = O.orderid
    WHERE O.custid = 1 ) AS D;

```

Табличные выражения упрощают поиск проблем, потому что вы всегда можете вычленить из кода команду `SELECT`, на которой они основаны, и выполнить ее без изменения исходных данных. В нашем примере табличные выражения используются для удобства. Но в некоторых ситуациях без них просто не обойтись. Чтобы это наглядно продемонстрировать, нам понадобится таблица `T1`, создать и заполнить которую можно посредством следующего кода.

```

IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
CREATE TABLE dbo.T1(col1 INT, col2 INT);
GO

INSERT INTO dbo.T1(col1) VALUES(10),(20),(30);

SELECT * FROM dbo.T1;

```

Содержимое таблицы `T1`, возвращаемое командой `SELECT`, показано ниже.

col1	col2
10	NULL
20	NULL
30	NULL

Представьте, что вам нужно обновить таблицу, присвоив столбцу `col2` результат выражения с функцией `ROW_NUMBER`. Проблема в том, что эту функцию нельзя использовать вместе с инструкцией `SET` в рамках команды `UPDATE`. Попробуйте запустить код, представленный ниже.

```

UPDATE dbo.T1
    SET col2 = ROW_NUMBER() OVER(ORDER BY col1);

```



Вы получите следующее сообщение об ошибке.

Сообщение 4108, уровень 15, состояние 1, строка 2  
Оконные функции могут использоваться только в предложениях SELECT  
или ORDER BY.

Чтобы обойти эту проблему, определим табличное выражение, которое возвращает два столбца: первый (**col2**) предназначен для обновления, а второй (назовем его **rownum**) будет основан на выражении с функцией ROW\_NUMBER. Внешняя команда UPDATE, направленная к табличному выражению, будет присваивать столбцу **rownum** содержимое **col2**. Вот как будет выглядеть наше решение при условии использования ОТВ.

```
WITH C AS
(
    SELECT col1, col2, ROW_NUMBER() OVER(ORDER BY col1) AS rownum
    FROM dbo.T1
)
UPDATE C
    SET col2 = rownum;
```

Выполните запрос к таблице, чтобы увидеть последствия обновления.

```
SELECT * FROM dbo.T1;
```

Вы получите следующий результат.

col1	col2
10	1
20	2
30	3

## Изменение данных с использованием параметров TOP и OFFSET-FETCH

SQL Server поддерживает использование параметра TOP непосредственно в командах INSERT, UPDATE, DELETE и MERGE. В этом случае изменение данных прекращается по достижении определенного количества или процентного соотношения строк. К сожалению, в отличие от SELECT все остальные команды не позволяют сортировать конечный результат с помощью инструкции ORDER BY, если используется параметр TOP. Фактически изменению подлежат те строки, которые по той или иной причине первыми попали на обработку.

Классическим примером использования параметра TOP для изменения данных является разбиение одной крупной операции (например, удаление большого числа строк) на несколько мелких.

В языке T-SQL есть еще один параметр с аналогичным действием — OFFSET-FETCH, который является частью инструкции ORDER BY. Однако его нельзя применять для изменения данных (по крайней мере, напрямую), поскольку соответствующие команды исключают упорядочение конечного результата.

Чтобы продемонстрировать процесс редактирования содержимого таблиц с использованием параметра TOP, нам придется создать и заполнить таблицу **dbo.Orders**. Код для этого представлен ниже.

```
IF OBJECT_ID('dbo.OrderDetails', 'U') IS NOT NULL DROP TABLE dbo.
OrderDetails;
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;

CREATE TABLE dbo.Orders
(
    orderid          INT          NOT NULL,
    custid           INT          NULL,
    empid            INT          NOT NULL,
    orderdate        DATETIME     NOT NULL,
    requireddate     DATETIME     NOT NULL,
    shippeddate       DATETIME     NULL,
    shipperid        INT          NOT NULL,
    freight          MONEY        NOT NULL
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname         NVARCHAR(40) NOT NULL,
    shipaddress      NVARCHAR(60) NOT NULL,
    shipcity         NVARCHAR(15) NOT NULL,
    shipregion       NVARCHAR(15) NULL,
    shippostalcode  NVARCHAR(10) NULL,
    shipcountry      NVARCHAR(15) NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);
GO

INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
```

В следующем примере мы удалим из таблицы **Orders** 50 строк, используя команду **DELETE** в сочетании с параметром TOP.

```
DELETE TOP(50) FROM dbo.Orders;
```

Инструкцию **ORDER BY** нельзя применять при изменении данных с помощью параметра TOP, поэтому наш запрос имеет одну существенную проблему — вы не можете заранее сказать, какие именно 50 строк будут удалены. Как правило, в их число входят те записи, которые первыми попадут на обработку. Это свидетельствует о том, что параметр TOP имеет довольно ограниченное применение при редактировании данных.

То же самое касается команд **UPDATE** и **INSERT**. К примеру, следующий код обновляет 50 строк из таблицы **Orders**, увеличивая значение атрибута **freight** на 10.

```
UPDATE TOP(50) dbo.Orders
SET freight += 10.00;
```

Опять же мы не знаем, какие строки должны обновиться; это будут первые 50 записей, к которым получит доступ SQL Server.

Конечно же, нас интересует, какие данные будут затронуты нашими действиями, и нам бы не хотелось, чтобы они были выбраны случайным образом. Эту проблему

можно обойти путем изменения содержимого таблиц через табличные выражения. Выражения, выполняющие запрос `SELECT`, могут содержать как параметр `TOP`, так и инструкцию `ORDER BY`, определяющую порядок следования строк. К результатам, полученным таким образом, можно будет применить команду, которая изменяет данные.

Например, следующий код удаляет 50 заказов с самым маленьким порядковым номером.

```
WITH C AS
(
    SELECT TOP(50) *
    FROM dbo.Orders
    ORDER BY orderid
)
DELETE FROM C;
```

Аналогичным образом код, представленный ниже, обновляет 50 заказов с самыми большими порядковыми номерами, увеличивая значение атрибута `freight` на 10.

```
WITH C AS
(
    SELECT TOP(50) *
    FROM dbo.Orders
    ORDER BY orderid DESC
)
UPDATE C
    SET freight += 10.00;
```

В SQL Server 2012 в рамках вложенных запросов `SELECT` вместо `TOP` можно использовать параметр `OFFSET-FETCH`. Вот видоизмененный пример с командой `DELETE`.

```
WITH C AS
(
    SELECT *
    FROM dbo.Orders
    ORDER BY orderid
    OFFSET 0 ROWS FETCH FIRST 50 ROWS ONLY
)
DELETE FROM C;
```

А это еще одна вариация примера с командой `UPDATE`.

```
WITH C AS
(
    SELECT *
    FROM dbo.Orders
    ORDER BY orderid DESC
    OFFSET 0 ROWS FETCH FIRST 50 ROWS ONLY
)
UPDATE C
    SET freight += 10.00;
```

## Инструкция OUTPUT

Обычно команды, предназначенные для изменения данных, не возвращают никакого результата. Однако в некоторых ситуациях возможность извлечения содержимого измененных строк оказывается полезной. Например, представьте, что команда UPDATE помимо обновления данных возвращает старые и новые значения измененных столбцов. Это могло бы пригодиться при поиске ошибок, проверке кода и т. д.

В SQL Server такая возможность реализована в виде инструкции OUTPUT, что добавляется к самой команде; внутри нее описываются атрибуты и выражения, которые нужно вернуть из отредактированных строк.

Инструкция OUTPUT во многом похожа на SELECT. Внутри нее тоже перечисляются элементы, основанные на имеющихся атрибутах. Отличие заключается в том, что имена всех атрибутов должны иметь один из двух префиксов — `inserted` или `deleted`. Первый используется при добавлении данных, а второй — при удалении. Что касается команды UPDATE, то префикс `deleted` позволяет получить содержимое строк до их изменения, а `inserted` — после.

Как и в инструкции SELECT, указанные атрибуты, извлекаемые из отредактированных строк, возвращаются в виде результирующего набора. Вы можете перенаправить результат прямо в таблицу, указав ее имя с помощью инструкции INTO. Если вы хотите, чтобы одна копия измененных данных записалась в таблицу, а другая вернулась в качестве результата, укажите две инструкции OUTPUT — с INTO и без.

В следующих разделах рассматриваются примеры использования инструкции OUTPUT в сочетании с разными командами для изменения данных.

### Сочетание с командой INSERT

Примером совместного использования команды INSERT и инструкции OUTPUT может служить ситуация, когда в таблицу со свойством `identity` вставляется набор строк и одновременно необходимо получить все значения, сгенерированные этим свойством. Функция `SCOPE_IDENTITY` возвращает только самое последнее значение, созданное в рамках сессии, поэтому в данном случае она бесполезна. Инструкция OUTPUT легко решает эту проблему. Чтобы продемонстрировать принцип ее работы, создадим таблицу T1 со столбцами `keycol` и `datacol`, первый из которых имеет свойство `identity`.

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;

CREATE TABLE dbo.T1
(
    keycol INT NOT NULL IDENTITY(1, 1) CONSTRAINT PK_T1 PRIMARY KEY,
    datacol NVARCHAR(40) NOT NULL
);
```

Допустим, вы хотите добавить в таблицу T1 результат выполнения запроса, направленного к таблице `HR.Employees`. Чтобы получить все значения `identity`, сгенерированные командой INSERT, достаточно просто добавить инструкцию OUTPUT и указать атрибуты, которые нужно вернуть.

keycol	datacol
-----	-----
1	Дэвис
2	Функ
3	Лью
4	Пелед
5	Камерон

(строк обработано: 5)

Как вы уже, наверное, догадались, подобный подход можно использовать для получения элементов последовательности, генерируемых внутри команды INSERT функцией NEXT VALUE FOR (либо напрямую, либо посредством ограничения DEFAULT).

Выше я уже упоминал, что результирующий набор можно направить прямо в таблицу (как в обычную, так и во временную) или табличную переменную. Результат, сохраненный таким образом, легко извлекать с помощью стандартных запросов. Например, следующий код объявляет табличную переменную @NewRows и направляет в нее результат работы инструкции OUTPUT, которая попутно вставляет результирующий набор в таблицу T1. В конце выполняется запрос, выводющий содержимое @NewRows.

```
DECLARE @NewRows TABLE(keycol INT, datacol NVARCHAR(40));

INSERT INTO dbo.T1(datacol)
    OUTPUT inserted.keycol, inserted.datacol
    INTO @NewRows
    SELECT lastname
    FROM HR.Employees
    WHERE country = N'Великобритания';

SELECT * FROM @NewRows;
```

В итоге мы получим следующий результат.

keycol	datacol
-----	-----
6	Бак
7	Суурс
8	Кинг
9	Долгопятова

(строк обработано: 4)

## Сочетание с командой DELETE

Ниже демонстрируется пример использования инструкции OUTPUT в сочетании с командой DELETE. Для начала запустите следующий код, чтобы создать копию таблицы Sales.Orders в схеме dbo.

```
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;

CREATE TABLE dbo.Orders
```

```
(
   orderid      INT          NOT NULL,
    custid      INT          NULL,
    empid       INT          NOT NULL,
    orderdate   DATETIME     NOT NULL,
    requireddate DATETIME     NOT NULL,
    shippeddate  DATETIME     NULL,
    shipperid   INT          NOT NULL,
    freight     MONEY        NOT NULL
    CONSTRAINT DFT_Orders_freight DEFAULT(0),
    shipname     NVARCHAR(40) NOT NULL,
    shipaddress  NVARCHAR(60) NOT NULL,
    shipcity     NVARCHAR(15) NOT NULL,
    shipregion   NVARCHAR(15) NULL,
    shippostalcode NVARCHAR(10) NULL,
    shipcountry  NVARCHAR(15) NOT NULL,
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)
);
GO

INSERT INTO dbo.Orders SELECT * FROM Sales.Orders;
```

Теперь удалим все заказы, размещенные до 2008 г. и с помощью инструкции OUTPUT извлечем атрибуты из удаленных строк.

```
DELETE FROM dbo.Orders
OUTPUT
    deleted.orderid,
    deleted.orderdate,
    deleted.empid,
    deleted.custid
WHERE orderdate < '20080101';
```

Команда DELETE вернет следующий результирующий набор.

orderid	orderdate	empid	custid
-----	-----	-----	-----
10248	2006-07-04 00:00:00.000	5	85
10249	2006-07-05 00:00:00.000	6	79
10250	2006-07-08 00:00:00.000	4	34
10251	2006-07-08 00:00:00.000	3	84
10252	2006-07-09 00:00:00.000	4	76
...			
10400	2007-01-01 00:00:00.000	1	19
10401	2007-01-01 00:00:00.000	1	65
10402	2007-01-02 00:00:00.000	8	20
10403	2007-01-03 00:00:00.000	4	20
10404	2007-01-03 00:00:00.000	2	49
...			
(строк обработано: 560)			

Если вы хотите заархивировать удаленные строки, просто добавьте инструкцию INTO и укажите имя таблицы, которая должна выступать в качестве архива.

## Сочетание с командой UPDATE

Используя инструкцию OUTPUT в сочетании с командой UPDATE, вы можете получить содержимое строк до изменения (добавляя к атрибутам префикс `deleted`) и после него (с помощью префикса `inserted`). Это позволяет возвращать как старые, так и новые значения обновляемых атрибутов.

Прежде чем переходить к примеру использования инструкции OUTPUT и команды UPDATE, нужно создать копию таблицы `Sales.OrderDetails` в схеме `dbo`. Для этого запустите следующий код.

```
IF OBJECT_ID('dbo.OrderDetails', 'U') IS NOT NULL DROP TABLE dbo.
OrderDetails;

CREATE TABLE dbo.OrderDetails
(
   orderid    INT           NOT NULL,
    productid INT           NOT NULL,
    unitprice MONEY        NOT NULL
    CONSTRAINT DFT_OrderDetails_unitprice DEFAULT(0),
    qty        SMALLINT     NOT NULL
    CONSTRAINT DFT_OrderDetails_qty    DEFAULT(1),
    discount   NUMERIC(4, 3) NOT NULL
    CONSTRAINT DFT_OrderDetails_discount DEFAULT(0),
    CONSTRAINT PK_OrderDetails PRIMARY KEY(orderid, productid),
    CONSTRAINT CHK_discount CHECK (discount BETWEEN 0 AND 1),
    CONSTRAINT CHK_qty CHECK (qty > 0),
    CONSTRAINT CHK_unitprice CHECK (unitprice >= 0)
);
GO

INSERT INTO dbo.OrderDetails SELECT * FROM Sales.OrderDetails;
```

Команда UPDATE, представленная ниже, увеличивает скидку для продукта под номером 51 на 5 %, затем, используя инструкцию OUTPUT, возвращает идентификатор продукта и старую/новую скидку для всех обновленных строк.

```
UPDATE dbo.OrderDetails
    SET discount += 0.05
OUTPUT
    inserted.productid,
    deleted.discount AS olddiscount,
    inserted.discount AS newdiscount
WHERE productid = 51;
```

Этот запрос возвращает следующий результат.

productid	olddiscount	newdiscount
51	0.000	0.050
51	0.150	0.200
51	0.100	0.150
51	0.200	0.250
51	0.000	0.050
51	0.150	0.200
51	0.000	0.050

```
51          0.000          0.050
51          0.000          0.050
51          0.000          0.050
...
(строк обработано: 39)
```

Сочетание с командой MERGE

Вы также можете использовать инструкцию OUTPUT в сочетании с командой MERGE, но помните; один вызов этой команды нередко состоит сразу из нескольких элементов языка DML, связанных условной логикой. Это означает, что инструкция OUTPUT может вернуть строки, полученные в результате выполнения разных команд. Чтобы определить происхождение той или иной строки, нужно воспользоваться функцией \$action, которая возвращает действие (INSERT, UPDATE или DELETE) в виде строки. Давайте обратимся к примеру, который мы рассматривали в разделе «Слияние данных». Но для начала необходимо заново запустить код из листинга 8.2, чтобы воссоздать таблицы dbo.Customers и dbo.CustomersStage.

Следующий код копирует содержимое CustomersStage в таблицу Customers, обновляя уже имеющиеся строки и добавляя новые.

```
MERGE INTO dbo.Customers AS TGT
USING dbo.CustomersStage AS SRC
  ON TGT.custid = SRC.custid
WHEN MATCHED THEN
  UPDATE SET
    TGT.companyname = SRC.companyname,
    TGT.phone = SRC.phone,
    TGT.address = SRC.address
WHEN NOT MATCHED THEN
  INSERT (custid, companyname, phone, address)
  VALUES (SRC.custid, SRC.companyname, SRC.phone, SRC.address)
OUTPUT $action AS theaction, inserted.custid,
       deleted.companyname AS oldcompanyname,
       inserted.companyname AS newcompanyname,
       deleted.phone AS oldphone,
       inserted.phone AS newphone,
       deleted.address AS oldaddress,
       inserted.address AS newaddress;
```

Здесь посредством инструкции OUTPUT возвращаются старые и новые значения измененных строк. Конечно, в случае выполнения команды INSERT не может быть никаких «старых значений», поэтому все ссылки на удаленные атрибуты будут содержать NULL. Функция \$action помогает узнать, какая команда вернула ту или иную строку — UPDATE или INSERT. Ниже представлен результат выполнения предыдущего примера.

theaction	custid	oldcompanyname	newcompanyname
UPDATE	2	AAAAA	AAAAA
UPDATE	3	cust 3	cust 3
UPDATE	5	BBBBB	BBBBB
UPDATE	6	cust 6 (new)	cust 6 (new)
UPDATE	7	cust 7 (new)	cust 7 (new)



ltheaction	custid	oldphone	newphone	oldaddress	newaddress
UPDATE	2	(222) 222-2222	(222) 222-2222	address 2	address 2
UPDATE	3	(333) 333-3333	(333) 333-3333	address 3	address 3
UPDATE	5	CCCCC	CCCCC	DDDDD	DDDDD
UPDATE	6	(666) 666-6666	(666) 666-6666	address 6	address 6
UPDATE	7	(777) 777-7777	(777) 777-7777	address 7	address 7

(строк обработано: 5)

## Компоновка элементов языка DML

Инструкция `OUTPUT` возвращает результат для каждой измененной строки. Но что, если вам нужно перенаправить в таблицу определенное подмножество строк (например, для дальнейшей проверки)? `SQL Server` поддерживает компоновку элементов языка `DML`, позволяя добавлять в целевую таблицу только нужные вам строки.

Чтобы продемонстрировать эту возможность, необходимо сначала скопировать таблицу `Production.Products` в схему `dbo` и создать новую таблицу под названием `dbo.ProductsAudit`.

```
IF OBJECT_ID('dbo.ProductsAudit', 'U') IS NOT NULL DROP TABLE dbo.
ProductsAudit;
IF OBJECT_ID('dbo.Products', 'U') IS NOT NULL DROP TABLE dbo.Products;

CREATE TABLE dbo.Products
(
    productid      INT          NOT NULL,
    productname    NVARCHAR(40) NOT NULL,
    supplierid     INT          NOT NULL,
    categoryid     INT          NOT NULL,
    unitprice      MONEY        NOT NULL
        CONSTRAINT DFT_Products_unitprice DEFAULT(0),
    discontinued   BIT          NOT NULL
        CONSTRAINT DFT_Products_discontinued DEFAULT(0),
    CONSTRAINT PK_Products PRIMARY KEY(productid),
    CONSTRAINT CHK_Products_unitprice CHECK(unitprice >= 0)
);

INSERT INTO dbo.Products SELECT * FROM Production.Products;

CREATE TABLE dbo.ProductsAudit
(
    LSN INT NOT NULL IDENTITY PRIMARY KEY,
    TS DATETIME NOT NULL DEFAULT(CURRENT_TIMESTAMP),
    productid INT NOT NULL,
    colname SYSNAME NOT NULL,
    oldval SQL_VARIANT NOT NULL,
    newval SQL_VARIANT NOT NULL
);
```

Теперь представьте, что вам нужно обновить все товары, отправленные поставщиком под номером 1, а именно увеличить их цену на 15 %. Вместе с тем необходимо

проверить старые и новые значения обновленных товаров, но только тех из них, у которых старая цена была меньше 20, а новая — больше или равна 20.

Для этого можно воспользоваться сочетанием разных элементов языка DML. Сначала на основе команды UPDATE с инструкцией OUTPUT определяется производная таблица. Затем с помощью команд INSERT и SELECT к этой таблице выполняется запрос, который отбирает только нужные нам строки. Ниже представлено полноценное решение.

```
INSERT INTO dbo.ProductsAudit(productid, colname, oldval, newval)
    SELECT productid, N'unitprice', oldval, newval
FROM (UPDATE dbo.Products
    SET unitprice *= 1.15
    OUTPUT
        inserted.productid,
        deleted.unitprice AS oldval,
        inserted.unitprice AS newval
    WHERE supplierid = 1) AS D
WHERE oldval < 20.0 AND newval >= 20.0;
```

При обсуждении логической обработки запросов и табличных выражений уже упоминалось, что мультимножество, возвращаемое запросом, может подаваться на вход следующим командам. Здесь результат выполнения инструкции OUTPUT (мультимножество) используется в качестве входящего значения для команды SELECT, результирующий набор которой в итоге вставляется в таблицу.

Запустите следующий код, чтобы вывести содержимое таблицы ProductsAudit.

```
SELECT * FROM dbo.ProductsAudit;
```

Результат показан ниже.

LSN	TS	productid	colname	oldval	newval
1	2014-03-09 12:41:01.043	1	unitprice	18,00	20,70
2	2014-03-09 12:41:01.043	2	unitprice	19,00	21,85

В итоге было обновлено три товара, но только два из них были отображены внешним запросом и скопированы в таблицу для дальнейшей проверки.

Закончив с примерами, очистите БД с помощью представленного ниже кода.

```
IF OBJECT_ID('dbo.OrderDetails', 'U') IS NOT NULL DROP TABLE dbo.
OrderDetails;
IF OBJECT_ID('dbo.ProductsAudit', 'U') IS NOT NULL DROP TABLE dbo.
ProductsAudit;
IF OBJECT_ID('dbo.Products', 'U') IS NOT NULL DROP TABLE dbo.Products;
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
IF OBJECT_ID('dbo.Customers', 'U') IS NOT NULL DROP TABLE dbo.Customers;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;
IF OBJECT_ID('dbo.Sequences', 'U') IS NOT NULL DROP TABLE dbo.Sequences;
IF OBJECT_ID('dbo.CustomersStage', 'U') IS NOT NULL DROP TABLE dbo.
CustomersStage;
```

## В заключение

В этой главе мы познакомились с множеством вопросов, касающихся изменения данных. Были описаны такие операции, как добавление, обновление, удаление и слияние. Мы также обсудили изменение данных с использованием табличных выражений и параметра TOP (уделив несколько абзацев параметру OFFSET-FETCH), а также научились возвращать измененные записи с помощью инструкции OUTPUT.

## Упражнения

В этом разделе представлены упражнения, которые помогут вам закрепить материал, пройденный в главе 8. Все примеры выполняются в контексте БД TSQL2012.

### 1

Запустите следующий код, чтобы создать в БД TSQL2012 таблицу `dbo.Customers`.

```
USE TSQL2012;

IF OBJECT_ID('dbo.Customers', 'U') IS NOT NULL DROP TABLE dbo.Customers;

CREATE TABLE dbo.Customers
(
    custid          INT          NOT NULL PRIMARY KEY,
    companyname     NVARCHAR(40) NOT NULL,
    country         NVARCHAR(15) NOT NULL,
    region         NVARCHAR(15) NULL,
    city           NVARCHAR(15) NOT NULL
);
```

### 1-1

Добавьте в таблицу `dbo.Customers` строку со следующими значениями:

- `custid`: 100
- `companyname`: Рога и копыта
- `country`: США
- `region`: WA
- `city`: Редмонд

### 1-2

Выберите из таблицы `Sales.Customers` записи о всех клиентах, которые размещали заказы, и скопируйте их в таблицу `dbo.Customers`.

## 1-3

С помощью команды `SELECT INTO` создайте таблицу `dbo.Orders` и заполните ее заказами из таблицы `Sales.Orders`, которые были размещены в 2006–2008 гг. Стоит отметить, что это упражнение можно выполнить только на локальной версии SQL Server, поскольку SQL Database не поддерживает команду `SELECT INTO` (вместо этого используются команды `CREATE TABLE` и `INSERT SELECT`).

## 2

Удалите из таблицы `dbo.Orders` заказы, размещенные до августа 2006 г. Используйте инструкцию `OUTPUT`, чтобы вернуть атрибуты `orderid` и `orderdate`, принадлежащие удаленным строкам.

- Ожидаемый результат:

orderid	orderdate
-----	-----
10248	2006-07-04 00:00:00.000
10249	2006-07-05 00:00:00.000
10250	2006-07-08 00:00:00.000
10251	2006-07-08 00:00:00.000
10252	2006-07-09 00:00:00.000
10253	2006-07-10 00:00:00.000
10254	2006-07-11 00:00:00.000
10255	2006-07-12 00:00:00.000
10256	2006-07-15 00:00:00.000
10257	2006-07-16 00:00:00.000
10258	2006-07-17 00:00:00.000
10259	2006-07-18 00:00:00.000
10260	2006-07-19 00:00:00.000
10261	2006-07-19 00:00:00.000
10262	2006-07-22 00:00:00.000
10263	2006-07-23 00:00:00.000
10264	2006-07-24 00:00:00.000
10265	2006-07-25 00:00:00.000
10266	2006-07-26 00:00:00.000
10267	2006-07-29 00:00:00.000
10268	2006-07-30 00:00:00.000
10269	2006-07-31 00:00:00.000

(строк обработано: 22)

## 3

Удалите из таблицы `dbo.Orders` заказы, размещенные бразильскими клиентами.

## 4

Выполните запрос к таблице `dbo.Customers`, представленный ниже. Обратите внимание, что некоторые строки содержат `NULL` в столбце `region`.

```
SELECT * FROM dbo.Customers;
```

Результат должен быть следующим.

custid	companyname	country	region	city
1	Клиент NRZBB	Германия	NULL	Берлин
2	Клиент MLTDN	Мексика	NULL	Мехико
3	Клиент KBUDE	Мексика	NULL	Мехико
4	Клиент HFBZG	Великобритания	NULL	Лондон
5	Клиент HGV LZ	Швеция	NULL	Лулео
6	Клиент XH XJV	Германия	NULL	Мангейм
7	Клиент QXVLA	Франция	NULL	Страсбург
8	Клиент QUHWH	Испания	NULL	Мадрид
9	Клиент RTXGC	Франция	NULL	Марсель
10	Клиент EEALV	Канада	BC	Тсаввассен
...				

(строка обработано: 90)

Теперь обновите таблицу `dbo.Customers`, заменяя отметки NULL значениями '`<None>`'. Используйте инструкцию OUTPUT, чтобы вывести содержимое столбцов `custid`, `oldregion` и `newregion`.

Ожидаемый результат:

custid	oldregion	newregion
1	NULL	<None>
2	NULL	<None>
3	NULL	<None>
4	NULL	<None>
5	NULL	<None>
6	NULL	<None>
7	NULL	<None>
8	NULL	<None>
9	NULL	<None>
11	NULL	<None>
12	NULL	<None>
13	NULL	<None>
14	NULL	<None>
16	NULL	<None>
17	NULL	<None>
18	NULL	<None>
19	NULL	<None>
20	NULL	<None>
23	NULL	<None>
24	NULL	<None>
25	NULL	<None>
26	NULL	<None>
27	NULL	<None>
28	NULL	<None>
29	NULL	<None>
30	NULL	<None>
39	NULL	<None>
40	NULL	<None>
41	NULL	<None>
44	NULL	<None>
49	NULL	<None>
50	NULL	<None>
52	NULL	<None>
53	NULL	<None>

54	NULL	<None>
56	NULL	<None>
58	NULL	<None>
59	NULL	<None>
60	NULL	<None>
63	NULL	<None>
64	NULL	<None>
66	NULL	<None>
68	NULL	<None>
69	NULL	<None>
70	NULL	<None>
72	NULL	<None>
73	NULL	<None>
74	NULL	<None>
76	NULL	<None>
79	NULL	<None>
80	NULL	<None>
83	NULL	<None>
84	NULL	<None>
85	NULL	<None>
86	NULL	<None>
87	NULL	<None>
90	NULL	<None>
91	NULL	<None>

(строк обработано: 58)

## 5

Обновите в таблице `dbo.Orders` все заказы, которые были размещены клиентами из Великобритании; присвойте их атрибутам `shipcountry`, `shipregion` и `shipcity` значения `country`, `region` и `city`, взятые из таблицы `dbo.Customers`.

Закончив с упражнениями, выполните следующий код, чтобы очистить БД.

```
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;
IF OBJECT_ID('dbo.Customers', 'U') IS NOT NULL DROP TABLE dbo.Customers;
```

## Решения

Здесь приводятся решения для упражнений, представленных в этой главе.

### 1-1

Убедитесь, что вы подключены к БД `TSQL2012`.

```
USE TSQL2012;
```

Чтобы добавить в таблицу `Customers` строку со значениями, перечисленными в условии, используйте команду `INSERT VALUES`.

```
INSERT INTO dbo.Customers(custid, companyname, country, region, city)
VALUES(100, N'Рога и копыта', N'США', N'WA', N'Редмонд');
```

```
    shippostalcode NVARCHAR(10) NULL,  
    shipcountry    NVARCHAR(15) NOT NULL,  
    CONSTRAINT PK_Orders PRIMARY KEY(orderid)  
);  
  
INSERT INTO dbo.Orders  
    (orderid, custid, empid, orderdate, requireddate, shippeddate,  
     shipperid, freight, shipname, shipaddress, shipcity, shipregion,  
     shippostalcode, shipcountry)  
SELECT  
    orderid, custid, empid, orderdate, requireddate, shippeddate,  
    shipperid, freight, shipname, shipaddress, shipcity, shipregion,  
    shippostalcode, shipcountry  
FROM Sales.Orders  
WHERE orderdate >= '20060101'  
      AND orderdate < '20090101';
```

## 2

Чтобы удалить заказы, размещенные до августа 2006 г., вам понадобятся команда DELETE и фильтр, основанный на предикате `orderdate < '20060801'`. Согласно условию вам нужно вернуть атрибуты из удаленных строк; для этого подойдет инструкция OUTPUT.

```
DELETE FROM dbo.Orders  
    OUTPUT deleted.orderid, deleted.orderdate  
WHERE orderdate < '20060801';
```

## 3

В данном упражнении от вас требуется написать команду DELETE, которая удаляет из таблицы `dbo.Orders` строки, имеющие соответствие в таблице `dbo.Customers`. Для решения этой задачи можно воспользоваться стандартной командой DELETE с предикатом EXISTS внутри инструкции WHERE, как показано ниже.

```
DELETE FROM dbo.Orders  
WHERE EXISTS  
    (SELECT *  
     FROM dbo.Customers AS C  
     WHERE Orders.custid = C.custid  
           AND C.country = N'Бразилия');
```

Этот код удаляет из таблицы `dbo.Orders` записи о клиентах, проживающих в Бразилии, идентификаторы которых хранятся в таблице `dbo.Customers`.

То же самое можно сделать посредством команды DELETE, основанной на соединении. Этот способ не является стандартным и поддерживается только в языке T-SQL.

```
DELETE FROM O  
FROM dbo.Orders AS O  
    JOIN dbo.Customers AS C  
        ON O.custid = C.custid  
WHERE country = N'Бразилия';
```

## 1-2

Чтобы определить, размещал ли клиент заказ, можно воспользоваться предикатом EXISTS, как показано ниже.

```
SELECT custid, companyname, country, region, city
FROM Sales.Customers AS C
WHERE EXISTS
  (SELECT * FROM Sales.Orders AS O
   WHERE O.custid = C.custid);
```

Чтобы добавить в таблицу **dbo.Customers** строки, которые вернул этот запрос, необходимо выполнить команду INSERT SELECT.

```
INSERT INTO dbo.Customers(custid, companyname, country, region, city)
  SELECT custid, companyname, country, region, city
  FROM Sales.Customers AS C
  WHERE EXISTS
    (SELECT * FROM Sales.Orders AS O
     WHERE O.custid = C.custid);
```

## 1-3

Сначала необходимо удостовериться в том, что текущая сессия подключена к БД **TSQL2012**. Затем с помощью команды SELECT INTO нужно удалить и заново создать таблицу **dbo.Orders**, копируя в нее заказы из таблицы **Sales.Orders**, которые были размещены в 2006–2008 гг.

```
USE TSQL2012;

IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;

SELECT *
INTO dbo.Orders
FROM Sales.Orders
WHERE orderdate >= '20060101'
  AND orderdate < '20090101';
```

Если вы работаете с SQL Database, нужно воспользоваться командами CREATE TABLE и INSERT SELECT.

```
CREATE TABLE dbo.Orders
(
 orderid          INT          NOT NULL,
  custid          INT          NULL,
  empid          INT          NOT NULL,
  orderdate       DATETIME    NOT NULL,
  requireddate    DATETIME    NOT NULL,
  shippeddate      DATETIME    NULL,
  shipperid       INT          NOT NULL,
  freight         MONEY       NOT NULL,
  shipname        NVARCHAR(40) NOT NULL,
  shipaddress     NVARCHAR(60) NOT NULL,
  shipcity        NVARCHAR(15) NOT NULL,
  shipregion      NVARCHAR(15) NULL,
```



Следует принять во внимание, что после выполнения первого запроса в таблице больше не осталось соответствий.

Соединение таблиц `dbo.Orders` и `dbo.Customers` играет роль фильтра, который сопоставляет заказы с клиентами. Инструкция `WHERE` отбирает тех клиентов, которые проживают в Бразилии. Команда `DELETE FROM` ссылается на псевдоним `O`, представляющий целевую таблицу `Orders`.

Есть еще один стандартный вариант решения данной задачи, который подразумевает использование команды `MERGE`. Обычно эта команда используется для выполнения разных действий на основе условной логики, но на самом деле не обязательно указывать больше одного действия. Иными словами, для выполнения слияния достаточно одной лишь инструкции `WHEN MATCHED`; условие `WHEN NOT MATCHED` можно пропустить. Код, представленный ниже, решает поставленную задачу.

```
MERGE INTO dbo.Orders AS O
USING dbo.Customers AS C
  ON O.custid = C.custid
  AND country = N'Бразилия'
WHEN MATCHED THEN DELETE;
```

Опять же нужно понимать, что после выполнения любого из вышеприведенных запросов в таблице не остается соответствий.

Команда `MERGE` определяет таблицу `dbo.Orders` как целевую, а сама таблица `dbo.Customers` используется в качестве источника. Из таблицы `dbo.Orders` удаляется информация о клиентах, которые проживают в Бразилии, чей идентификатор содержится в источнике.

## 4

Это упражнение подразумевает написание команды `UPDATE`, которая отбирает строки с атрибутом `region`, равным `NULL`. Для поиска неопределенных значений нужно использовать предикат `IS NULL`, а не оператор равенства. Вернуть запрашиваемую информацию можно, прибегнув к инструкции `OUTPUT`. Полноценное решение представлено ниже.

```
UPDATE dbo.Customers
  SET region = '<None>'
OUTPUT
  deleted.custid,
  deleted.region AS oldregion,
  inserted.region AS newregion
WHERE region IS NULL;
```

## 5

Эту задачу легко решить с помощью нестандартной версии команды `UPDATE`, основанной на соединении. Таблицы `dbo.Orders` и `dbo.Customers` соединяются при условии соответствия клиентских идентификаторов, которые в них хранятся. Инструкция `WHERE` будет отбирать только тех клиентов, которые проживают в Великобритании. В инструкции `UPDATE` нужно указать псевдоним, который будет

представлять обновляемую таблицу **dbo.Orders**. Инструкция **SET** позволяет указать место доставки, которое будет совпадать с адресом соответствующего клиента. Полноценное решение представлено ниже.

```
UPDATE O
  SET shipcountry = C.country,
      shipregion = C.region,
      shipcity = C.city
FROM dbo.Orders AS O
  JOIN dbo.Customers AS C
    ON O.custid = C.custid
WHERE C.country = 'Великобритания';
```

Еще один вариант решения заключается в использовании ОТВ. Вы можете определить его на основе запроса **SELECT**, который соединяет таблицы **dbo.Orders** и **dbo.Customers**, возвращая исходное и целевое местоположения. Затем команда **UPDATE** во внешнем запросе заменяет целевые атрибуты исходными значениями. Итоговый код будет выглядеть так.

```
WITH CTE_UPD AS
(
  SELECT
    O.shipcountry AS ocountry, C.country AS ccountry,
    O.shipregion AS oregion, C.region AS cregion,
    O.shipcity AS ocity, C.city AS ccity
  FROM dbo.Orders AS O
    JOIN dbo.Customers AS C
      ON O.custid = C.custid
  WHERE C.country = 'Великобритания'
)
UPDATE CTE_UPD
  SET ocountry = ccountry, oregion = cregion, ocity = ccity;
```

Того же результата можно добиться посредством команды **MERGE**. Как уже объяснялось ранее, в этой команде обычно используются две инструкции — **WHEN MATCHED** и **WHEN NOT MATCHED**, но никто нам не запрещает ограничиться только одной из них. Инструкция **WHEN MATCHED** в сочетании с операцией обновления представляет собой логический эквивалент первых двух решений. Вот как это должно выглядеть.

```
MERGE INTO dbo.Orders AS O
  USING dbo.Customers AS C
    ON O.custid = C.custid
    AND C.country = 'Великобритания'
  WHEN MATCHED THEN
    UPDATE SET shipcountry = C.country,
               shipregion = C.region,
               shipcity = C.city;
```

## Глава 9

# ТРАНЗАКЦИИ И ПАРАЛЛЕЛИЗМ

Глава посвящена работе транзакций. Вы узнаете, каким образом Microsoft SQL Server обрабатывает параллельные запросы, принадлежащие разным пользователям, но обращенные к одним и тем же данным. Я расскажу, как с помощью блокировок изолировать несогласованные данные, решать проблемы с блокированием и менять степень согласованности при запросах к данным, используя разные уровни изоляции. Мы также рассмотрим ситуации с взаимным блокированием и научимся минимизировать риск их появления.

## Транзакции

**Транзакция** — это некая единица работы, которая может включать различные операции, извлекающие и изменяющие как сами данные, так и их определение.

Транзакцию можно обозначить явно или неявно. Для ее открытия используется инструкция `BEGIN TRAN` (или `BEGIN TRANSACTION`). Чтобы подтвердить выполнение транзакции, нужно указать инструкцию `COMMIT TRAN`; если вы не хотите ее подтверждать (в этом случае все изменения будут отменены), используйте инструкцию `ROLLBACK TRAN` (или `ROLLBACK TRANSACTION`). Ниже показан пример создания транзакции, которая выполняет команды `INSERT`.

```
BEGIN TRAN;
    INSERT INTO dbo.T1(keycol, col1, col2) VALUES(4, 101, 'C');
    INSERT INTO dbo.T2(keycol, col1, col2) VALUES(4, 201, 'X');
COMMIT TRAN;
```

В SQL Server каждая отдельная команда считается транзакцией и подтверждается по умолчанию. Вы можете изменить это поведение, используя параметр сессии под названием `IMPLICIT_TRANSACTIONS` (изначально выключен), который требует явного обозначения конца транзакции посредством инструкций `COMMIT TRAN` или `ROLLBACK TRAN`; хотя начало по-прежнему можно опускать.

Транзакции характеризуются четырьмя свойствами — атомарностью, согласованностью, изоляцией и надежностью, которые принято обозначать аббревиатурой ACID (atomicity, consistency, isolation, durability).

- **Атомарность.** Транзакция — это атомарная единица работы. Она либо полностью выполняет все запланированные действия, либо не делает ничего. Если до ее завершения (то есть до того, как инструкция `COMMIT TRAN` будет записана в журнал) произошел какой-то системный сбой, после перезапуска SQL Server отменит все изменения. В случае возникновения обычных ошибок транзакция автоматически откатывается обратно. Бывают ситуации, когда автоматиче-

ская отмена результатов транзакции считается неоправданной, например нарушение условий первичного ключа или истечение времени блокирования (подробней об этом мы поговорим в разделе «Проблемы, связанные с блокированием»). Такие ошибки легко перехватывать с помощью кода обработки, принимая все необходимые меры (например, вы можете записать ошибку в журнал и откатить транзакцию). Эта тема рассматривается в главе 10.



#### СОВЕТ

В SQL Server есть функция @@TRANSCOUNT. Если текущий участок кода выполняется внутри транзакции, она возвращает положительное число. В противном случае возвращается 0.

- **Согласованность.** Таким термином обозначают состояние данных, к которым предоставляется параллельный доступ на чтение и запись. Как вы понимаете, это довольно субъективное понятие, суть которого зависит от нужд приложения. Степень согласованности, установленная по умолчанию в SQL Server, рассматривается в разделе «Уровни изоляции»; там же вы узнаете, каким образом ее можно изменить. Под согласованностью понимают еще и тот факт, что БД должна соблюдать все аспекты целостности, определенные внутри нее с помощью различных правил, таких как первичные и внешние ключи, ограничения уникальности и т. д. В результате транзакции база данных меняет одно согласованное состояние на другое.
- **Изоляция.** Этот механизм используется для управления доступом к данным. Он следит за тем, чтобы транзакция получала доступ только к той информации, которая обладает необходимым уровнем согласованности. SQL Server поддерживает два вида изоляции: традиционный, основанный на блокировании, и новый, который подразумевает управление версиями строк. Первый используется по умолчанию в локальных версиях SQL Server и при выполнении чтения требует наличия разделяемых блокировок. Чтение блокируется до тех пор, пока состояние данных не станет согласованным. Второй вид изоляции, основанный на управлении версиями строк, используется по умолчанию в Windows Azure SQL Database. Он снимает необходимость в разделяемых блокировках и позволяет клиентам производить чтение без задержек. Если текущее состояние данных не согласовано, клиент получает более старую версию, которая годится к употреблению. Подробную информацию об этих двух подходах можно найти в разделе «Уровни изоляции».
- **Надежность.** Прежде чем сохранять данные непосредственно в БД, SQL Server всегда записывает изменения в журнал транзакций. Если в него попала инструкция COMMIT TRAN, транзакция считается надежной; при этом нет никакой гарантии, что данные действительно были записаны на диск. При запуске (не важно, в обычном режиме или после сбоя) система сверяется с журналом транзакций каждой БД и запускает процесс восстановления, который состоит из двух этапов — повторного выполнения и отмены. Первый этап подразумевает повторение всех действий, принадлежащих транзакциям, которые были зафиксированы в журнале, но не закончили процесс записи. На втором этапе выполняется откат (отмена) всех изменений, произведенных в рамках транзакций, которые не были подтверждены в журнале с помощью инструкции COMMIT TRAN.

Следующий код определяет транзакцию, которая записывает в БД TSQL2012 сведения о новом заказе.

```
USE TSQL2012;

-- Начинаем новую транзакцию
BEGIN TRAN;

-- Объявляем переменную
DECLARE @neworderid AS INT;

-- Добавляем новый заказ в таблицу Sales.Orders
INSERT INTO Sales.Orders
    (custid, empid, orderdate, requireddate, shippeddate,
     shipperid, freight, shipname, shipaddress, shipcity,
     shippostalcode, shipcountry)
VALUES
    (85, 5, '20090212', '20090301', '20090216',
     3, 32.38, N'Адрес: 85-В', N'6789 Рю де Л''Абе', N'Реймс',
     N'10345', N'Франция');

-- Сохраняем в переменной идентификатор нового заказа
SET @neworderid = SCOPE_IDENTITY();

-- Возвращаем идентификатор нового заказа
SELECT @neworderid AS neworderid;

-- Добавляем данные о заказанном товаре в таблицу Sales.OrderDetails
INSERT INTO Sales.OrderDetails
    (orderid, productid, unitprice, qty, discount)
VALUES(@neworderid, 11, 14.00, 12, 0.000),
    (@neworderid, 42, 9.80, 10, 0.000),
    (@neworderid, 72, 34.80, 5, 0.000);

-- Подтверждаем транзакцию
COMMIT TRAN;
```

Эта транзакция добавляет информацию о заказе в таблицу **Sales.Orders** и несколько строк со сведениями о заказанном товаре в таблицу **Sales.OrderDetails**. Идентификатор нового заказа генерируется автоматически, поскольку столбец **orderid** имеет свойство **identity**. Сразу после выполнения записи в таблицу **Sales.Orders** идентификатор присваивается локальной переменной и затем используется для добавления данных о заказанном товаре. В целях тестирования я добавил команду **SELECT**, которая выводит содержимое переменной (см. ниже).

```
neworderid
-----
11078
```

Стоит отметить, что данный пример не предусматривает обработки ошибок и отката транзакции в случае сбоя. Подробней об этом мы поговорим в главе 10. Там же вы познакомитесь с конструкцией **TRY/CATCH**.

Закончив с примерами, очистите БД с помощью следующего кода.

```
DELETE FROM Sales.OrderDetails  
WHERE orderid > 11077;
```

```
DELETE FROM Sales.Orders  
WHERE orderid > 11077;
```

## Блокировки и блокирование

Для обеспечения изоляции транзакций в SQL Server используются блокировки. В следующих разделах речь пойдет о процессе блокирования и том, как избежать ситуаций с конфликтующими блокирующими запросами.

### Блокировки

Блокировки управляют ресурсами, чтобы защитить данные и предотвратить несовместимость и конфликты, вызванные работой других транзакций. Сначала вы познакомитесь с наиболее важными режимами блокировок, которые поддерживаются в SQL Server, и узнаете, каким образом они между собой совмещаются. Затем я расскажу о типах ресурсов, которые могут быть заблокированы.

### Режимы и совместимость

При изучении транзакций и параллелизма первым делом нужно обратить внимание на два режима блокировок — монопольный и разделяемый.

При изменении данных транзакция запрашивает монопольную блокировку на ресурсы, с которыми она должна работать, вне зависимости от текущего уровня изоляции (об этом мы поговорим чуть позже). Если запрос был удовлетворен, блокировка удерживается до окончания транзакции. Если транзакция содержит несколько команд, блокирование прекратится только после выполнения инструкций `COMMIT TRAN` или `ROLLBACK TRAN`.

**Монопольной** называется блокировка, которая может быть получена только одной транзакцией; вы не можете ее получить, если ресурс уже заблокирован (не важно, в каком режиме). К тому же, если вы установили монопольную блокировку, другие транзакции не смогут выполнить какое-либо блокирование того же ресурса. Данный принцип действует по умолчанию, и его нельзя изменить, по крайней мере те его аспекты, которые касаются продолжительности (до завершения транзакции) и типа блокировки, необходимого для редактирования ресурса (монопольного). С практической точки зрения это означает, что строки, редактируемые одной транзакцией, доступны для всех остальных транзакций только в режиме чтения (и то при условии, что выставлен подходящий уровень изоляции).

Что касается операций чтения, то стандартные настройки SQL Server и SQL Database имеют различия. В локальном варианте по умолчанию используется уровень изоляции под названием `READ COMMITTED`. Он работает следующим образом: когда вы пытаетесь прочитать данные, ваша транзакция запрашивает разделяемую блокировку, которая снимается только после окончания чтения. Этот режим блокирования называется **разделяемым**, потому что он открывает доступ к одному и тому

же ресурсу одновременно из нескольких транзакций. И хотя во время записи будет нельзя поменять режим блокировки и ее продолжительность, у вас есть возможность влиять на процесс блокирования при выполнении чтения — именно для этого были созданы уровни изоляции. Подробно на данную тему мы поговорим чуть позже.

В SQL Database стандартным уровнем изоляции является `READ COMMITTED SNAPSHOT`. Вместо того чтобы полагаться на блокировки, он основывается на управлении версиями строк. Это означает, что для чтения данных не нужно выполнять разделяемое блокирование и, как следствие, не приходится ждать; расширенная изоляция обеспечивается наличием нескольких вариантов каждой отдельной строки. С практической точки зрения уровень изоляции `READ COMMITTED` не позволяет считывать данные, которые изменяются в рамках другой транзакции. Такой способ управления параллелизмом называют **пессимистическим**. Но есть и **оптимистический** подход, который реализован за счет уровня изоляции `READ COMMITTED SNAPSHOT`: если транзакция попытается прочесть строки, которые в это самое время подлежат изменению, она получит их последнее подтвержденное состояние, актуальное на момент запуска самой транзакции.

Взаимодействие между транзакциями, работающими с одними и теми же данными, называется **совместимостью блокировок**. В таблице 9.1 проиллюстрирована совместимость монопольных и разделяемых блокировок, сгенерированных на одном уровне изоляции. По горизонтали показаны режимы, которые запрашиваются, а по вертикали — те, которые уже установлены.

**Таблица 9.1.** Совместимость монопольных и разделяемых блокировок

Запрашиваемый режим	Установленный монопольный режим (X)	Установленный разделяемый режим (S)
Удовлетворить запрос о монопольной блокировке?	Нет	Нет
Удовлетворить запрос об эксклюзивной блокировке?	Нет	Да

Отрицательный ответ означает, что блокировки несовместимы и запрос об установлении режима будет отклонен, то есть запрашивающая сторона должна подождать. Положительный ответ указывает на совместимость блокировок и то, что запрошенный режим будет установлен.

Взаимодействие транзакций на уровне блокировок можно подытожить так: данные, которые изменяются одной транзакцией, не могут изменяться или считываться другой, пока первая не будет завершена. Если данные считываются одной транзакцией, они не могут быть изменены другой. По крайней мере, так по умолчанию ведут себя локальные версии SQL Server.

## Поддерживающие типы ресурсов

SQL Server может блокировать разные типы ресурсов, в том числе ключи, строковые идентификаторы, страницы, объекты (например, таблицы), БД и т. д. Строки находятся внутри страниц, которые представляют собой физические блоки информации, содержащие таблицы и индексные данные. Это те типы, на которые стоит обратить внимание в первую очередь. Позже мы обсудим другие виды ресур-

сов, поддерживающие блокирование, — экстенды, единицы распределения, кучи и сбалансированные деревья.

Чтобы заблокировать ресурс определенного типа, ваша транзакция должна получить намеренную блокировку того же режима, но на более высоком уровне детализации. Например, чтобы заблокировать строку в монопольном режиме, транзакция сначала должна получить намеренную монопольную блокировку страницы, в которой находится эта строка, и объекта, который хранит страницу. Точно такой же принцип действует для разделяемых блокировок. Намеренное блокирование позволяет определять и отклонять запросы на получение несовместимых блокировок на более высоком уровне детализации. Например, если одна транзакция блокирует строку, а другая запрашивает несовместимую блокировку для страницы или целой таблицы, в которой находится эта строка, SQL Server легко обнаруживает конфликт, поскольку он знает о намеренной блокировке первой транзакции, установленной для страницы и таблицы. Намеренные блокировки не конфликтуют с запросами на более низком уровне детализации. Например, намеренное блокирование страницы не мешает другим транзакциям получать блокировки в совместимом режиме для строк, которые там находятся. Ниже представлено дополнение к таблице 9.1, в котором представлена информация о совместимости намеренных монопольных и намеренных разделяемых блокировок.

**Таблица 9.2.** Совместимость разных видов блокировок, включая намеренные

Запрашиваемый режим	Установленный монопольный режим (X)	Установленный разделяемый режим (S)	Установленный монопольный режим (IX)	Установленный разделяемый режим (IS)
Удовлетворить запрос о монопольной блокировке?	Нет	Нет	Нет	Нет
Удовлетворить запрос об эксклюзивной блокировке?	Нет	Да	Нет	Да
Удовлетворить запрос о намеренной монопольной блокировке?	Нет	Нет	Да	Да
Удовлетворить запрос о намеренной эксклюзивной блокировке?	Нет	Да	Да	Да

Типы блокируемых ресурсов определяются динамически. Естественно, в идеале блокировки должны устанавливаться только для нужных вам данных, то есть для тех строк, с которыми вы работаете. При блокировании выделяются дополнительная оперативная память и другие системные ресурсы — SQL Server все это учитывает.

Сначала SQL Server блокирует мелкий ресурс (строку или страницу), затем в некоторых случаях пытается заблокировать более крупный объект (такой как таблица). К примеру, это происходит в ситуации, когда одна команда получает как минимум 5000 блокировок; в случае неудачи попытка повторяется при установлении следующих 1250 блокировок.

В SQL Server 2008 и SQL Server 2012, используя команду `ALTER TABLE` можно установить параметр `LOCK_ESCALATION`, который позволяет управлять распространением блокировки. Например, ее можно отключить или ограничить на уровне



таблицы (что и происходит по умолчанию) или раздела (одна таблица может состоять из нескольких более мелких разделов).

## Проблемы, связанные с блокированием

Когда одна транзакция блокирует ресурс, а другая пытается установить для него несовместимую блокировку, запрос приостанавливается и входит в состояние ожидания. По умолчанию ожидание происходит до тех пор, пока первая транзакция не закончит блокирование ресурса. Позже я объясню, каким образом можно указать время ожидания для текущей сессии, чтобы приостановка транзакции не длилась дольше положенного времени.

Блокирование — это нормальное явление, но иногда оно слишком затягивается. В таких ситуациях стоит проанализировать поведение своих запросов и подумать, можно ли как-то избежать подобных задержек. Например, продолжительные транзакции, которые приводят к неоправданному увеличению времени ожидания, можно урезать. Для этого стоит вынести из них те операции, которые могут выполняться во внешнем коде. Случаются ошибки, из-за которых транзакции в определенных условиях остаются открытыми даже после завершения своей работы. Вам следует отслеживать подобные ошибки и делать так, чтобы транзакции закрывались всегда, независимо от условий.

В этом разделе рассматривается ситуация, в которой происходит блокирование, и даются пошаговые советы относительно того, как ее избежать. Я исхожу из того, что вы используете локальную версию SQL Server и уровень изоляции `READ COMMITTED`; это означает, что команды `SELECT` по умолчанию запрашивают разделяемую блокировку. Напомню, что стандартным уровнем изоляции в SQL Database является `READ COMMITTED SNAPSHOT`, в котором при выборке данных разделяемая блокировка не запрашивается.

Если вы хотите выполнить пример, представленный ниже, в контексте SQL Database, вам придется добавить к команде `SELECT` табличное указание `READCOMMITTEDLOCK`, допустим, `SELECT * FROM T1 WITH (READCOMMITTEDLOCK)`. Кроме того, в SQL Database по умолчанию используется довольно короткое время ожидания соединения, поэтому код, демонстрируемый здесь, может работать не так, как вы того ожидаете.

Откройте в SQL Server Management Studio три отдельные вкладки для выполнения запросов (я буду называть их **Соединение 1**, **Соединение 2** и **Соединение 3**). Убедитесь в том, что все они подключены к БД `TSQL2012`.

```
USE TSQL2012;
```

Запустите следующий код в контексте **Соединения 1**, чтобы обновить строку в таблице `Production.Products`, а именно добавить 1,00 к текущей цене товара под номером 2 (которая изначально равна 19,00).

```
BEGIN TRAN;

UPDATE Production.Products
SET unitprice += 1.00
WHERE productid = 2;
```

Чтобы иметь возможность вносить изменения в строку, ваша сессия должна запросить монопольную блокировку; если изменения пройдут успешно, SQL Server удовлетворит этот запрос. Как вы помните, монопольные блокировки удерживаются до завершения транзакции. Но поскольку наша транзакция не закрыта, блокировка все еще остается на месте.

Попробуйте получить ту же строку, запустив следующий код в контексте **Соединения 2** (если вы работаете с SQL Database, в этом и последующих запросах нужно убрать символы «--» во второй строке).

```
SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;
```

Чтобы прочитать эти данные, вашей сессии нужна разделяемая блокировка. Поскольку строка уже монопольно блокируется другой сессией, а монопольные и разделяемые блокировки несовместимы между собой, вашей транзакции придется остановиться и подождать.

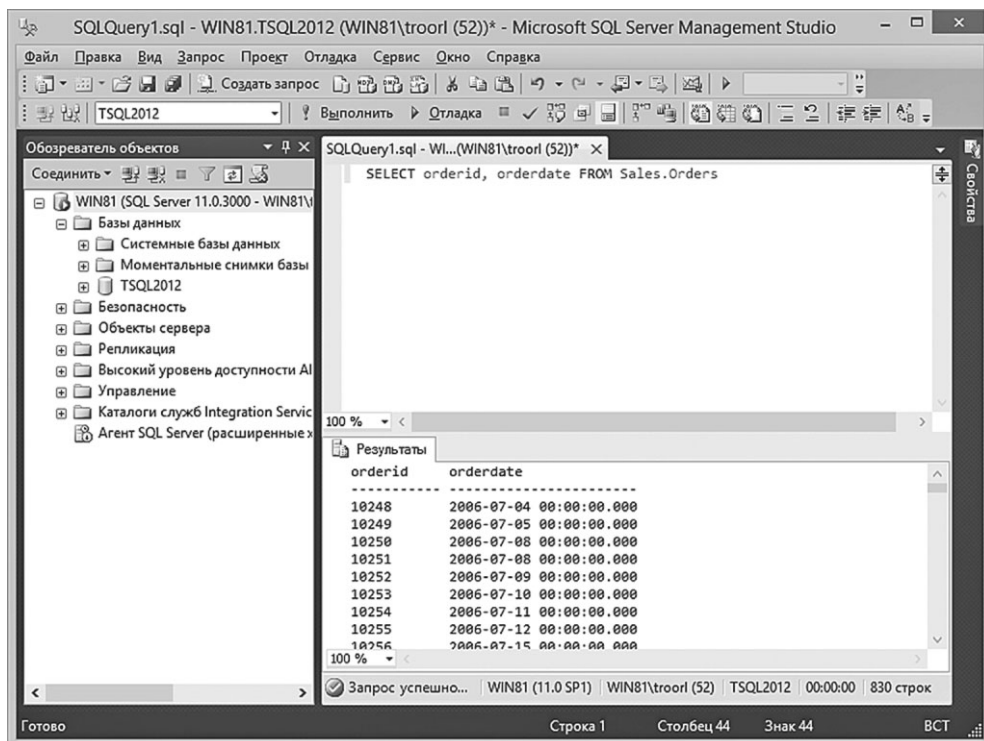
Если предположить, что подобные ситуации станут периодически возникать в системе, а время ожидания заблокированной сессии начнет существенно затягиваться, вы, вероятно, захотите как-то решить возникшую проблему. В этом вам помогут запросы к динамическим управляющим объектам, таким как представления и функции, которые нужно будет запускать в контексте **Соединения 3**.

Чтобы получить информацию о блокировках (как текущих, так и тех, выдача которых все еще ожидается), перейдите в **Соединение 3** и обратитесь к динамическому управляющему представлению (dynamic management view, или DMV) под названием `sys.dm_tran_locks`.

```
SELECT -- используйте *, чтобы вывести все доступные атрибуты
    request_session_id      AS spid,
    resource_type            AS restype,
    resource_database_id     AS dbid,
    DB_NAME(resource_database_id) AS dbname,
    resource_description     AS res,
    resource_associated_entity_id AS resid,
    request_mode             AS mode,
    request_status           AS status
FROM sys.dm_tran_locks;
```

Я запустил этот код на своей локальной системе (без других открытых вкладок) и получил следующий результат.

spid	restype	dbid	dbname	res	resid	mode	status
53	DATABASE	8	TSQL2012		0	S	GRANT
52	DATABASE	8	TSQL2012		0	S	GRANT
51	DATABASE	8	TSQL2012		0	S	GRANT
54	DATABASE	8	TSQL2012		0	S	GRANT
53	PAGE	8	TSQL2012	1:127	72057594038845440	IS	GRANT
52	PAGE	8	TSQL2012	1:127	72057594038845440	IX	GRANT
53	OBJECT	8	TSQL2012		133575514	IS	GRANT
52	OBJECT	8	TSQL2012		133575514	IX	GRANT
52	KEY	8	TSQL2012	(020068e8b274)	72057594038845440	X	GRANT
53	KEY	8	TSQL2012	(020068e8b274)	72057594038845440	S	WAIT



**Рис. 9.1.** Идентификатор SPID в окне SQL Server Management Studio

Каждой сессии назначается уникальный идентификатор серверного процесса (server process ID, или SPID), узнать который можно, используя функцию @@SPID. В SQL Server Management Studio значение SPID выводится внизу окна, в строке состояния (справа от имени пользователя), а также в заголовке вкладки, внутри которой выполняются запросы. Например, на рисунке 9.1 показано окно SQL Server Management Studio, в котором идентификатор 54 находится справа от логина WIN81/troorl.

В результате запроса, направленного к представлению `sys.dm_tran_locks`, было найдено четыре сессии, которые удерживают блокировки (51, 52, 53 и 54). Вы можете наблюдать следующие атрибуты:

- тип заблокированного ресурса (например, KEY в случае строки индекса);
- идентификатор БД, в которой выполняется блокирование. Вы можете транслировать его в имя БД, используя функцию DB\_NAME;
- ресурс и его идентификатор;
- режим блокировки;
- состояние блокировки (уже выдана или еще ожидается).

Я советую вам не ограничиваться этими атрибутами и изучить всю информацию о блокировках, которая доступна в данном представлении.

Все сведения, содержащиеся в представлении `sys.dm_tran_locks`, касаются только процессов, вовлеченных в цепочку блокировок. Чтобы получить информацию о соединениях, связанных с этими процессами, нужно выполнить запрос к представлению `sys.dm_exec_connections` и отобразить соответствующие идентификаторы `SPID`.

Стоит отметить, что в цепочке блокировок участвовали процессы с идентификаторами 52 и 53. Эти данные актуальны только для моей системы — у вас могут быть другие результаты. Когда будете запускать запросы, приведенные выше, не забудьте оставить идентификаторы, которые используются в вашей цепочке блокировок.

```

spid      connect_time      last_read
-----
52        2012-06-25 15:20:03.360    2012-06-25 15:20:15.750
53        2012-06-25 15:20:07.300    2012-06-25 15:20:20.950

last_write      most_recent_sql_handle
-----
2012-06-25 15:20:15.817 0x01000800DE2B71FB0936F0500000000000000000000000000000000
2012-06-25 15:20:07.327 0x02000000063FC7D052E098447778CDD615CFE7A2D1FB411802

```

- время их установления;
- время, когда они в последний раз производили чтение или запись;
- двоичное значение, хранящее дескриптор последнего пакета, который был выполнен в рамках соединения. Данный дескриптор необходимо передать в табличную функцию под названием `sys.dm_exec_sql_text`, которая возвращает пакет с соответствующим кодом внутри. Это можно сделать

вручную, но более удобным решением является использование табличного оператора `APPLY`, с которым мы познакомились в главе 5. Он применяет функцию к каждой строке, представляющей соединение, как показано ниже (запустите этот код в контексте **Соединения 3**).

```
SELECT session_id, text
FROM sys.dm_exec_connections
    CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) AS ST
WHERE session_id IN(52, 53);
```

Результат, который я получил, содержит пакеты с последним кодом, выполненным в рамках соединений, являющихся частью цепочки блокировок.

```
session_id  text
-----
52          BEGIN TRAN;

            UPDATE Production.Products
               SET unitprice += 1.00
               WHERE productid = 2;

53          (@! tinyint)
            SELECT [productid],[unitprice]
            FROM [Production].[Products]
            WHERE [productid]=@1
```

Последний запрос, запущенный в рамках заблокированного процесса (53), находится в состоянии ожидания. Вы можете также видеть код, приведший к блокированию, — он принадлежит процессу под номером 52. Однако нужно понимать, что этот код может все еще работать, поэтому причиной проблемы не обязательно является самая последняя команда.

Множество полезной информации относительно сессий, участвующих в блокировании, можно найти в динамическом управляющем представлении под названием `sys.dm_exec_sessions`. Запрос, показанный ниже, возвращает лишь небольшое подмножество доступных атрибутов.

```
SELECT -- используйте *, чтобы вывести все доступные атрибуты
    session_id AS spid,
    login_time,
    host_name,
    program_name,
    login_name,
    nt_user_name,
    last_request_start_time,
    last_request_end_time
FROM sys.dm_exec_sessions
WHERE session_id IN(52, 53);
```

Полученный результат пришлось разбить на несколько частей.

```
spid login_time                host_name
----
52    2012-06-25 15:20:03.407  K2
53    2012-06-25 15:20:07.303  K2
```

spid	program_name	login_name
52	Microsoft SQL Server Management Studio - Query	WIN81/troorl
53	Microsoft SQL Server Management Studio - Query	WIN81/troorl

spid	nt_user_name	last_request_start_time	last_request_end_time
52	troorl	2012-06-25 15:20:15.703	2012-06-25 15:20:15.750
53	troorl	2012-06-25 15:20:20.693	2012-06-25 15:20:07.320

Данный результирующий набор содержит такую информацию, как время входа в систему, имя компьютера, название программы, логин, имя пользователя в операционной системе, время запуска и завершения последнего запроса. Эти сведения могут дать вам исчерпывающее представление о том, чем занимаются ваши сессии.

`sys.dm_exec_requests` — еще одно динамическое управляющее представление, которое может оказаться крайне полезным при решении проблем с блокированием. Оно содержит информацию обо всех активных запросах, включая блокирующие. Фактически вы легко можете извлечь только те из них, которые занимаются получением блокировок; для этого достаточно отобрать строки, в которых атрибут `blocking_session_id` больше нуля. Ниже показан пример.

```
SELECT -- используйте *, чтобы вывести все доступные атрибуты
    session_id AS spid,
    blocking_session_id,
    command,
    sql_handle,
    database_id,
    wait_type,
    wait_time,
    wait_resource
FROM sys.dm_exec_requests
WHERE blocking_session_id > 0;
```

Полученный результат разбит на несколько частей.

spid	blocking_session_id	command
53	52	SELECT

spid	sql_handle	database_id
53	0x0200000063FC7D052E09844778CDD615CFE7A2D1FB411802	8

spid	wait_type	wait_time	wait_resource
53	LCK_M_S	1383760	KEY: 8:72057594038845440 (020068e8b274)

Вы легко можете определить, какие сессии участвуют в цепочке блокирования, какие ресурсы в это вовлечены, как долго заблокированная сессия находится в состоянии ожидания (в миллисекундах) и т. д.

Если вам необходимо прервать процесс, вызвавший блокирование (например, если в результате программной ошибки транзакция остается открытой и само

приложение неспособно ее закрыть), вы можете воспользоваться командой `KILL <spid>` (но пока не спешите это делать). Стоит отметить, что на момент написания этой книги данная команда все еще не доступна в SQL Database.

Ранее я уже упоминал, что по умолчанию сессии не имеют времени ожидания блокирования. Если вы хотите ограничить время, отведенное на ожидание получения блокировки, можете использовать параметр `LOCK_TIMEOUT`. Значение задается в миллисекундах, например 5000 (5 секунд), 0 (нулевое) или -1 (неограниченное).

Чтобы увидеть этот параметр в действии, остановите сначала **Соединение 1**, выбрав пункт **Отменить выполнение запроса** в меню **Запрос (Alt+Break)**. Затем запустите следующий код, чтобы установить время ожидания блокировки (5 секунд), и потом сразу же сделайте повторный запуск.

```
SET LOCK_TIMEOUT 5000;

SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;
```

Запрос остается заблокированным, поскольку **Соединение 1** все еще не завершило выполнение транзакции. Если просьба о выдаче блокировки не будет удовлетворена, SQL Server прервет запрос — и вы получите следующее сообщение.

```
Сообщение 1222, уровень 16, состояние 51, строка 3
Блокировки время ожидания запроса превышено.
```

Стоит отметить, что данная ошибка не приводит к откату транзакции.

Чтобы сделать время ожидания неограниченным, верните параметру `LOCK_TIMEOUT` значение по умолчанию (-1) и попробуйте снова запустить свой код, на этот раз в контексте **Соединения 2**.

```
SET LOCK_TIMEOUT -1;

SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;
```

Чтобы вручную прервать выполнение транзакции в **Соединении 1**, запустите следующий код в контексте **Соединения 3** (я исхожу из того, что вы работаете с локальной версией SQL Server).

```
KILL 52;
```

Данная команда приводит к откату транзакции в **Соединении 1**. Это означает, что цена продукта под номером 2 вернется к своему исходному значению (19,00), а монополярная блокировка будет снята. Чтобы в этом убедиться, перейдите к **Соединению 2** — как видите, данные получены уже после отмены обновления (то есть до изменения цены).

```
productid  unitprice
-----
2          19.00
```

## Уровни изоляции

Уровни изоляции определяют поведение пользователей, которые считывают или записывают данные в один и тот же момент времени. Чтение по умолчанию подразумевает выборку данных с использованием разделяемых блокировок. Запись же требует монопольной блокировки, так как она приводит к изменению содержимого таблицы. Способом получения и продолжительностью действия блокировок можно управлять только при чтении. На запись тоже можно влиять, но косвенно; это делается с помощью уровней изоляции, которые выставляются либо посредством параметра сессии, либо в самом запросе в виде табличного указания.

SQL Server поддерживает четыре традиционных уровня изоляции, основанных на пессимистическом управлении параллелизмом (блокировании): `READ UNCOMMITTED`, `READ COMMITTED` (используется по умолчанию в локальных версиях SQL Server), `REPEATABLE READ` и `SERIALIZABLE`. Поддерживаются также два дополнительных уровня, в основе которых лежит оптимистичный подход к параллелизму (управление версиями строк): `SNAPSHOT` и `READ COMMITTED SNAPSHOT` (используется по умолчанию в SQL Database); они являются «оптимистичными» аналогами уровней `READ COMMITTED` и `SERIALIZABLE` соответственно.

В некоторых статьях `READ COMMITTED` и `READ COMMITTED SNAPSHOT` рассматриваются как один и тот же уровень, только с разной семантикой.

Уровень изоляции можно распространить на всю сессию, используя следующую команду.

```
SET TRANSACTION ISOLATION LEVEL <название уровня изоляции>;
```

Но можно ограничиться только текущим запросом.

```
SELECT ... FROM <table> WITH (<название_уровня_изоляции >);
```

Стоит отметить, что в первом случае название уровня изоляции должно содержать пробелы, если оно состоит из более чем одного слова (как в случае с `REPEATABLE READ`). Табличное указание требует записывать названия слитно, например `WITH (REPEATABLE READ)`; при этом вы можете использовать специальные псевдонимы: `NOLOCK` для `READ UNCOMMITTED`, `HOLDLOCK` для `SERIALIZABLE` и т.д.

Стандартный уровень изоляции в локальных версиях SQL Server, `READ COMMITTED`, основан на блокировании. В SQL Database по умолчанию используется уровень `READ COMMITTED SNAPSHOT` (на основе управления версиями строк). Меняя эти режимы, вы можете влиять не только на параллельный доступ к БД, но и на согласованность ее содержимого.

Если взять первые четыре уровня, то чем они выше, тем более строгие блокировки запрашиваются при чтении и тем длиннее время их ожидания; вместе с уровнем повышается согласованность, но ухудшаются условия параллельной работы. Обратное утверждение тоже верно.

С двумя другими уровнями изоляции все обстоит не так. SQL Server способен хранить ранее подтвержденные версии строк в БД `tempdb`. При чтении пользователи



не запрашивают разделяемые блокировки; вместо этого они пытаются получить более старую версию записи, если текущая им не подходит.

В следующих разделах будут рассмотрены все шесть уровней изоляции и то, как они себя ведут.

## READ UNCOMMITTED

READ UNCOMMITTED — наиболее низкий уровень изоляции, который не требует получения разделяемой блокировки для чтения данных. Тем самым исключается конфликт с пользователями, которые производят запись в монопольном режиме. Это означает, что вы можете считывать неподтвержденные данные (данный процесс называется **грязным чтением**). Пока операции чтения происходит на уровне изоляции READ UNCOMMITTED, пользователи могут свободно записывать данные.

Чтобы увидеть, как происходит чтение неподтвержденных данных, откройте две вкладки для выполнения запросов (я буду называть их **Соединение 1** и **Соединение 2**). Убедитесь, что все действия происходят в контексте БД **TSQL2012**.

Запустите в рамках **Соединения 1** код, представленный ниже; он откроет транзакцию, увеличит цену товара под номером 2 на 1,00 (до 20,00) и затем запросит измененную строку.

```
BEGIN TRAN;

UPDATE Production.Products
    SET unitprice += 1.00
WHERE productid = 2;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Следует отметить, что транзакция осталась открытой — строка с товаром монопольно блокируется **Соединением 1**. После выполнения вышеприведенного кода мы получим запись с новой ценой.

productid	unitprice
2	20.00

Перейдите к **Соединению 2** и запустите следующий код, который устанавливает режим изоляции READ UNCOMMITTED и запрашивает строку с товаром под номером 2.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Поскольку этот запрос не пытается получить разделяемую блокировку, он не конфликтует с другой транзакцией. Мы получим строку в том виде, который она приняла после обновления, несмотря на то что изменения не были подтверждены.

productid	unitprice
-----	-----
2	20.00

Не стоит забывать, что **Соединение 1** может и дальше редактировать эту строку в рамках той же транзакции или, например, откатить все изменения, как показано ниже.

```
ROLLBACK TRAN;
```

Данная команда отменяет обновление продукта под номером 2, возвращая старую цену (19,00). Значение 20,00, которое мы получили при чтении, так и не было подтверждено. Это пример грязного чтения.

## READ COMMITTED

Если вы хотите запретить чтение неподтвержденных изменений, вам нужен более строгий уровень изоляции. Самым низким уровнем, который позволяет это сделать, является `READ COMMITTED`; он используется по умолчанию в локальных версиях SQL Server. Как видно из названия, `READ COMMITTED` допускает чтение только тех изменений, которые были подтверждены. Для всего остального требуется получить разделяемую блокировку, которая конфликтует с монопольной. Это означает, что если данные уже изменяются, пользователь, которому нужно их прочитать, должен будет подождать. Он получит разделяемую блокировку сразу, как только транзакция подтвердит изменения.

Чтобы это продемонстрировать, перейдите к **Соединению 1** и выполните следующий код, который обновляет цену товара под номером 2 и запрашивает измененную строку с новой ценой.

```
BEGIN TRAN;

UPDATE Production.Products
SET unitprice += 1.00
WHERE productid = 2;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Результат показан ниже.

productid	unitprice
-----	-----
2	20.00

Теперь **Соединение 1** монопольно блокирует строку с идентификатором 2.

Далее представлен код, который нужно запустить в контексте **Соединения 2**; он устанавливает `READ COMMITTED` в качестве уровня изоляции и запрашивает строку с товаром под номером 2 (в SQL Database нужно открыть комментарий, чтобы указать уровень `READCOMMITTEDLOCK`).

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;
```

Не забывайте, что данный уровень изоляции используется по умолчанию, поэтому его можно не указывать вручную (разве что вы его уже поменяли для всей сессии). Команда `SELECT` заблокирована, поскольку для выполнения чтения ей нужно получить разделяемую блокировку, которая конфликтует с монопольной блокировкой в [Соединении 1](#).

Теперь запустите в контексте [Соединения 1](#) код, представленный ниже, чтобы подтвердить транзакцию.

```
COMMIT TRAN;
```

Переключившись на вкладку с [Соединением 2](#), вы увидите следующий результат.

productid	unitprice
2	20.00

В отличие от `READ UNCOMMITTED`, уровень изоляции `READ COMMITTED` исключает грязное чтение. При нем разделяемая блокировка снимается сразу, как только заканчивается чтение ресурса, не дожидаясь закрытия транзакции; фактически для этого даже не нужно ждать, когда выполнится команда. Это означает, что если одна транзакция производит две операции чтения одного и того же ресурса, ей не нужно ничего блокировать. Следовательно, ресурс может быть изменен между данными операциями, в результате чего получится два разных значения. Это так называемое **неповторяющееся чтение** (или несогласованный анализ). Для некоторых приложений подобное явление вполне приемлемо.

Закончив с примерами, верните таблицу к исходному состоянию, запустив следующий код в любой открытой вкладке.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

## REPEATABLE READ

Если вы хотите запретить изменение данных между операциями чтения, которые проводятся в одной транзакции, вам следует повысить уровень изоляции до `REPEATABLE READ`. Таким образом, для чтения необходимо будет получить разделяемую блокировку, которая удерживается до завершения транзакции; это сделает невозможным монопольное блокирование и, как следствие, изменение ресурса. Тем самым вы обеспечите **повторяющееся чтение** (или согласованный анализ).

В следующем примере демонстрируется работа в режиме `REPEATABLE READ`. Запустите этот код в контексте [Соединения 1](#), чтобы установить уровень изоляции, открыть транзакцию и прочитать строку с товаром под номером 2.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
  
BEGIN TRAN;  
  
    SELECT productid, unitprice  
    FROM Production.Products  
    WHERE productid = 2;
```

В результате вы получите текущую цену соответствующего товара.

productid	unitprice
2	19.00

Поскольку мы работаем в режиме REPEATABLE READ, Соединение 1 будет удерживать разделяемую блокировку для строки с идентификатором 2 до окончания транзакции. Попробуйте изменить эту строку, запустив следующий код в контексте Соединения 2.

```
UPDATE Production.Products  
    SET unitprice += 1.00  
WHERE productid = 2;
```

Попытка записи будет отклонена, поскольку запрос на монопольное блокирование конфликтует с разделяемой блокировкой, которая была выдана для чтения. Если бы мы работали на уровнях изоляции READ UNCOMMITTED или READ COMMITTED, у нас бы не было разделяемой блокировки и попытка записи оказалась бы успешной.

Вернитесь к Соединению 1 и запустите следующий код, который повторно считывает строку с товаром под номером 2 и подтверждает транзакцию.

```
SELECT productid, unitprice  
FROM Production.Products  
WHERE productid = 2;  
  
COMMIT TRAN;
```

Результат показан ниже.

productid	unitprice
2	19.00

Стоит отметить, что при второй попытке чтения мы получим такую же цену, как и при первой. Теперь, когда транзакция завершена и разделяемая блокировка снята, Соединение 2 может, наконец, монопольно заблокировать и обновить строку.

Еще одно явление, которого можно избежать благодаря уровню изоляции REPEATABLE READ (в отличие от более низких уровней), называется **потерянным обновлением**. Представьте, что две транзакции считывают некое значение, производят на его основе вычисления и затем пытаются его обновить. Поскольку первые два уровня изоляции не удерживают блокировку после выполнения операции чтения, обновления смогут быть выполнены обеими транзакциями; причем «выигрывает» та из них, которая делает обновление последней, перезаписывая

предыдущие изменения. В режиме `REPEATABLE READ` обе стороны удерживают разделяемые блокировки до самого конца, поэтому ни одна из них не сможет позже заблокировать ресурс для выполнения записи. В результате получится взаимное блокирование, которое позволит избежать конфликта при обновлении. Подробнее об этом мы поговорим позже в разделе «Взаимное блокирование».

Закончив с примерами, верните таблицу к исходному состоянию, выполнив следующий код.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

## SERIALIZABLE

Уровень изоляции `REPEATABLE READ` приводит к тому, что разделяемая блокировка, полученная для чтения данных, удерживается до конца транзакции. Таким образом, на протяжении всей транзакции вы можете считывать одни и те же значения. При этом строки, которых не было при запуске запроса, не блокируются. Следовательно, они могут появиться при повторном чтении. Такие строки называются **фантомными**, а операции, которые их возвращают, — **фантомным чтением**. Подобная ситуация возникает, если между двумя обращениями к таблице происходит добавление новых строк в рамках другой транзакции; при этом добавленные строки попадают в результирующий набор одного из запросов первой транзакции.

Чтобы избежать фантомного чтения, необходимо перейти на уровень изоляции `SERIALIZABLE`. Этот режим во многом похож на `REPEATABLE READ` — в частности, для выполнения операций чтения он требует установления разделяемой блокировки, которая удерживается до конца транзакции. Но есть и отличие: такой уровень изоляции заставляет клиента, который считывает данные, блокировать весь диапазон ключей, соответствующих фильтру запроса. Это означает, что чтение блокирует все строки, с которыми будет происходить работа, как уже имеющиеся, так и те, что могут быть добавлены в будущем. Если быть более точным, данный режим не дает другим транзакциям добавлять строки, которые могут пройти фильтр запроса.

Проиллюстрируем это на реальном примере. Запустите следующий код в контексте **Соединения 1**, чтобы установить уровень изоляции `SERIALIZABLE`, открыть транзакцию и извлечь весь товар категории 1.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRAN

SELECT productid, productname, categoryid, unitprice
FROM Production.Products
WHERE categoryid = 1;
```

Как видно по результату, в категории 1 находится 12 товаров.

productid	productname	categoryid	unitprice
1	Продукт HHYDP	1	18,00
2	Продукт RECZE	1	19,00

24	Продукт	QOGNU	1	4,50
34	Продукт	SWNJY	1	14,00
35	Продукт	NEVTJ	1	18,00
38	Продукт	QDOMO	1	263,50
39	Продукт	LSOFL	1	18,00
43	Продукт	ZZZHR	1	46,00
67	Продукт	XLXQF	1	14,00
70	Продукт	TOONT	1	15,00
75	Продукт	BWRLG	1	7,75
76	Продукт	JYGFE	1	18,00

(строк обработано: 12)

Теперь попытайтесь добавить новый товар в категорию 1, выполнив следующий код в контексте **Соединения 2**.

```
INSERT INTO Production.Products
(productname, supplierid, categoryid,
unitprice, discontinued)
VALUES('Продукт ABCDE', 1, 1, 20.00, 0);
```

На первых трех уровнях изоляции эта попытка была бы успешной, но в режиме **SERIALIZABLE** данный код приведет к блокированию.

Вернитесь к **Соединению 1** и запустите запрос, который повторно извлекает товары первой категории и завершает транзакцию.

```
SELECT productid, productname, categoryid, unitprice
FROM Production.Products
WHERE categoryid = 1;
```

```
COMMIT TRAN;
```

Вы получите тот же результат, что и прежде, но без фантомного чтения. Теперь, когда первая транзакция подтверждена и разделяемая блокировка диапазона ключей снята, **Соединение 2** может установить монопольную блокировку, разрешение на которую оно ожидало все это время, и начать добавление строк.

Закончив с примерами, очистите БД с помощью следующего кода.

```
DELETE FROM Production.Products
WHERE productid > 77;
```

Запустите во всех открытых соединениях код, представленный ниже, чтобы вернуть стандартный уровень изоляции.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

## Уровни изоляции, основанные на управлении версиями строк

SQL Server позволяет хранить предыдущие версии подтвержденных строк в БД tempdb. Этот механизм лежит в основе двух уровней изоляции: **SNAPSHOT** и **READ COMMITTED SNAPSHOT**. Первый из них похож на режим **SERIALIZABLE** —

по крайней мере с точки зрения проблем согласованности, которые он может или не может решить; второй напоминает `READ COMMITTED`. Тем не менее эти два уровня не предусматривают разделяемых блокировок при чтении, поэтому они не заставляют ждать снятия монопольной блокировки. Пользователи, выполняющие чтение, все равно могут рассчитывать на степень согласованности данных, которую предлагают уровни изоляции `SERIALIZABLE` и `READ COMMITTED`. Если в своем текущем состоянии строка не подходит для считывания, SQL Server предлагает прочитать более старую ее версию.

Нужно отметить, что при включении одного из этих двух уровней изоляции (в SQL Database они включены по умолчанию) команды `DELETE` и `UPDATE` перед внесением изменений будут копировать содержимое строк в БД `tempdb`; это не относится к команде `INSERT`, поскольку у строки, которая добавляется, не может быть предыдущих версий. Вы должны понимать, что уровни изоляции, основанные на управлении версиями строк, замедляют операции обновления и удаления. Производительность чтения данных, как правило, возрастает, поскольку исчезает необходимость в разделяемом блокировании и ожидании снятия монопольных блокировок. В следующих разделах рассматриваются теоретическая и практическая стороны уровней изоляции, в основе которых лежит создание снимков данных.

## SNAPSHOT

Уровень изоляции `SNAPSHOT` гарантирует, что клиент, считывающий данные, получит последнюю версию строки, подтвержденную на момент начала транзакции. Это обеспечивает повторяющееся и исключает фантомное чтение — точно так же, как в режиме `SERIALIZABLE`. При этом вместо разделяемых блокировок используется разбиение строк на версии. Как я уже упоминал, уровни изоляции, основанные на снимках данных, приводят к снижению производительности, особенно во время операций обновления и удаления (при этом не важно, включен ли один из этих режимов в текущей сессии). Таким образом, чтобы включить уровень изоляции `SNAPSHOT` в локальном экземпляре SQL Server (в SQL Database он уже включен по умолчанию), вам сначала необходимо активировать параметр `ALLOW_SNAPSHOT_ISOLATION` в контексте всей БД. Для этого откройте любую вкладку и запустите код, представленный ниже.

```
ALTER DATABASE TSQ12012 SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Чтобы продемонстрировать поведение режима `SNAPSHOT`, рассмотрим конкретный пример. Запустите в контексте **Соединения 1** следующий код, который открывает транзакцию, увеличивает цену товара под номером 2 на 1,00 (с 19,00) и запрашивает измененную строку с новой ценой.

```
BEGIN TRAN;

UPDATE Production.Products
    SET unitprice += 1.00
    WHERE productid = 2;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Ниже показан результат, согласно которому цена увеличилась до 20,00.

productid	unitprice
2	20.00

Стоит отметить, что перед изменением строки ее текущее состояние (с ценой 19,00) будет скопировано в БД `tempdb` даже в том случае, если **Соединение 1** работает в режиме `READ COMMITTED`. Так происходит потому, что уровень изоляции `SNAPSHOT` действует для всей БД. Однако этот уровень позволяет запрашивать предыдущую версию строки до обновления. Откройте **Соединение 2** и запустите в нем следующий код, который устанавливает режим `SNAPSHOT`, инициирует транзакцию и извлекает строку с товаром под номером 2.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

```
BEGIN TRAN;
```

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Если бы ваша транзакция работала на уровне изоляции `SERIALIZABLE`, запрос был бы заблокирован. У нас установлен режим `SNAPSHOT`, поэтому вы получили последнюю версию строки, которая была доступна на момент запуска транзакции. Эта версия (с ценой 19,00) не является текущей (поскольку цена уже успела обновиться до 20,00), поэтому SQL Server убирает ее из БД `tempdb`. Результат показан ниже.

productid	unitprice
2	19.00

Вернитесь к **Соединению 1** и подтвердите транзакцию, которая изменила строку.

```
COMMIT TRAN;
```

На этом этапе текущая версия строки (с ценой 20,00) уже подтверждена. Попытавшись прочитать данные в контексте **Соединения 2**, вы все равно получите ту версию, которая была актуальна на момент начала транзакции (ту, что с ценой 19,00). Чтобы в этом убедиться, запустите следующий код.

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

```
COMMIT TRAN;
```

Как и ожидалось, мы получили старую цену.

productid	unitprice
2	19.00



Вернитесь к **Соединению 2** и запустите код, представленный ниже, чтобы открыть и подтвердить новую транзакцию, попутно извлекая данные.

```
BEGIN TRAN

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;

COMMIT TRAN;
```

На этот раз последняя подтвержденная версия строки (с ценой 20,00) доступна на момент запуска транзакции. Поэтому результат будет следующим.

productid	unitprice
2	20.00

Версия с предыдущей ценой потеряла актуальность. Она будет удалена специальным потоком, который ежеминутно проводит очистку БД tempdb.

Закончив с примерами, приведите таблицу **Products** к исходному состоянию.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

## Обнаружение конфликтов

Уровень изоляции SNAPSHOT предотвращает некорректное обновление данных, но в отличие от уровней REPEATABLE READ и SERIALIZABLE, которые делают это за счет взаимных блокировок, он прерывает транзакцию, сигнализируя о наличии конфликта. Конфликт обнаруживается посредством анализа БД tempdb. В режиме SNAPSHOT SQL Server может определить, успели ли измениться данные между операциями чтения и записи, которые проводились в вашей транзакции.

Ниже приводятся два примера. В первом обновление проходит без проблем, а во втором возникает конфликт.

Запустите в рамках **Соединения 1** следующий код, который устанавливает уровень изоляции SNAPSHOT, открывает транзакцию и считывает строку с товаром под номером 2.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Результат представлен ниже.

productid	unitprice
2	19.00

Допустим, вы уже провели некие вычисления на основе прочитанных вами данных. Теперь попробуйте запустить в рамках того же соединения второй пример, который повышает цену ранее полученного товара до 20,00 и подтверждает транзакцию.

```
UPDATE Production.Products
    SET unitprice = 20.00
WHERE productid = 2;

COMMIT TRAN;
```

Пока производились чтение, вычисления и запись данных, ни одна другая транзакция не изменила вашу строку. В итоге не произошло никакого конфликта, поэтому SQL Server разрешил выполнить обновление.

Верните обратно исходную цену товара под номером 2.

```
UPDATE Production.Products
    SET unitprice = 19.00
WHERE productid = 2;
```

Запустите в рамках **Соединения 1** код, который снова открывает транзакцию и считывает строку с тем же товаром.

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN;

    SELECT productid, unitprice
    FROM Production.Products
    WHERE productid = 2;
```

Как видно по результату, цена товара равна 19,00.

productid	unitprice
2	19.00

Теперь перейдите к **Соединению 2** и запустите следующий код, чтобы повысить цену продукта под номером 2 до 25,00.

```
UPDATE Production.Products
    SET unitprice = 25.00
WHERE productid = 2;
```

Предположим, что в рамках **Соединения 1** были произведены некие вычисления, основанные на ранее прочитанной цене (19,00). Исходя из этого, попробуйте установить для товара под номером 2 цену 20,00.

```
UPDATE Production.Products
    SET unitprice = 20.00
WHERE productid = 2;
```

На этот раз SQL Server увидит, что между операциями чтения и записи работала другая транзакция, которая изменила ваши данные; следовательно, транзакция, инициированная **Соединением 1**, завершится следующей ошибкой.

Сообщение 3960, уровень 16, состояние 2, строка 1

Транзакция в режиме изоляции моментального снимка прервана из-за конфликта обновлений. Невозможно использовать режим изоляции моментального снимка для прямого или косвенного доступа к таблице "Production.Products" в базе данных "TSQL2012" для обновления, удаления или вставки строки, которая изменена или удалена другой транзакцией. Повторите транзакцию или измените уровень изоляции для инструкции обновления или удаления.

Конечно, при обнаружении конфликта можно обработать код ошибки и повторить всю транзакцию заново.

Закончив с примерами, верните таблицу **Products** в исходное состояние.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

Закройте все соединения. Если этого не сделать, результаты выполнения следующих примеров могут не совпасть с теми, которые приводятся в этой главе.

## READ COMMITTED SNAPSHOT

Уровень изоляции **READ COMMITTED SNAPSHOT** тоже основан на управлении версиями строк. В отличие от **SNAPSHOT** он предоставляет последний снимок данных, подтвержденный на момент запуска команды, а не транзакции. Кроме того, он не следит за возникновением конфликтов при операциях обновления. По принципу своей работы похож на уровень **READ COMMITTED**, если не считать того факта, что при нем чтение может производиться без получения разделяемых блокировок и без ожидания, связанного с монопольным блокированием.

Чтобы включить уровень изоляции **READ COMMITTED SNAPSHOT** в локальной версии SQL Server (в SQL Database он уже включен по умолчанию), вам необходимо активировать параметр БД **READ\_COMMITTED\_SNAPSHOT**. Продемонстрируем это на примере БД **TSQL2012**.

```
ALTER DATABASE TSQL2012 SET READ_COMMITTED_SNAPSHOT ON;
```

Стоит отметить, что для успешной работы данного кода БД **TSQL2012** должна иметь только одно открытое соединение.

Интересно, что с включением такого параметра уровень изоляции **READ COMMITTED** автоматически меняется на **READ COMMITTED SNAPSHOT**. Чтобы этого избежать, можно вручную установить режим для текущей сессии.

Чтобы увидеть в действии уровень изоляции **READ COMMITTED SNAPSHOT**, откройте два соединения. Выполните в контексте первого из них следующий код, который открывает транзакцию, обновляет данные о товаре под номером 2, считает эти данные и оставляет транзакцию открытой.

```
USE TSQL2012;

BEGIN TRAN;

UPDATE Production.Products
SET unitprice += 1.00
```

```
WHERE productid = 2;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Результат, представленный ниже, говорит о том, что цена была повышена до 20,00.

productid	unitprice
2	20.00

Перейдите к **Соединению 2**, откройте транзакцию и прочитайте строку с товаром под номером 2 (транзакция не должна закрываться).

```
BEGIN TRAN;

SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;
```

Вы получите последнюю подтвержденную версию строки, которая была доступна на момент запуска команды (19,00).

productid	unitprice
2	19.00

Теперь подтвердите транзакцию в **Соединении 1**.

```
COMMIT TRAN;
```

Перейдите к **Соединению 2**, повторно прочитайте строку с товаром под номером 2 и опять подтвердите транзакцию.

```
SELECT productid, unitprice
FROM Production.Products
WHERE productid = 2;

COMMIT TRAN;
```

Если бы код работал в режиме SNAPSHOT, вы бы получили цену 19,00; но поскольку определен уровень изоляции READ COMMITTED SNAPSHOT, вернулась последняя подтвержденная версия строки, доступная на момент запуска команды, а не транзакции.

productid	unitprice
2	20.00

Как вы помните, это явление называется неповторяющимся чтением (или несогласованным анализом).

Закончив с примерами, верните таблицу **Products** в исходное состояние.

```
UPDATE Production.Products
SET unitprice = 19.00
WHERE productid = 2;
```

Закройте все соединения. Если вы запускали этот код на локальной версии SQL Server, вам следует отключить в БД TSQ2012 все уровни изоляции, основанные на управлении версиями строк.

```
ALTER DATABASE TSQ2012 SET ALLOW_SNAPSHOT_ISOLATION OFF;
ALTER DATABASE TSQ2012 SET READ_COMMITTED_SNAPSHOT OFF;
```

## Краткий обзор всех уровней изоляции

В таблице 9.3 приводятся сведения о проблемах с согласованностью, которые могут или не могут возникнуть на каждом из уровней изоляции; в ней также проиллюстрированы поддержка обнаружения конфликтов при обновлении данных и возможность управления версиями строк.

**Таблица 9.3.** Сводка всех уровней изоляции

Уровень изоляции	Грязное чтение	Неповторяющееся чтение	Потерянные обновления	Фантомное чтение	Отслеживание конфликтов при обновлении?	Управление версиями строк
READ UNCOMMITTED	Да	Да	Да	Да	Нет	Нет
READ COMMITTED	Нет	Да	Да	Да	Нет	Нет
READ COMMITTED SNAPSHOT	Нет	Да	Да	Да	Нет	Да
REPEATABLE READ	Нет	Нет	Нет	Да	Нет	Нет
SERIALIZABLE	Нет	Нет	Нет	Нет	Нет	Нет
SNAPSHOT	Нет	Нет	Нет	Нет	Да	Да

## Взаимное блокирование

**Взаимное блокирование** возникает в ситуациях, когда два или больше процессов блокируют друг друга. Например, процесс А блокирует процесс Б, а процесс Б блокирует процесс А. Или процесс А блокирует процесс Б, процесс Б блокирует процесс В, а процесс В блокирует процесс А. Как бы то ни было, SQL Server отслеживает такие случаи и пытается их предотвратить, прерывая выполнение одной из транзакций. Если во взаимное блокирование не вмешаться извне, оно будет продолжаться вечно.

По умолчанию SQL Server останавливает ту транзакцию, которая последней проявляла какую-либо активность, потому что ее легче откатить к исходному состоянию. Чтобы повлиять на данный процесс, сессии можно назначить отдельный приоритет; для этого существует параметр DEADLOCK\_PRIORITY, который поддерживает 21 значение (от -10 до 10). Процесс с самым низким приоритетом будет первым в очереди на преждевременное завершение, независимо от того, сколько он успел проработать; в случае равенства приоритетов все будет решаться исходя из того, кто последним проявил активность.

Сначала мы рассмотрим простой пример взаимного блокирования, а затем поговорим о том, как уменьшить вероятность возникновения подобных ситуаций.

Откройте два соединения, направленных к БД **TSQL2012**. Выполните в контексте **Соединения 1** следующий код, который инициирует транзакцию и обновляет в таблице **Production.Products** строку с товаром под номером 2 (транзакция при этом не закрывается).

```
USE TSQL2012;

BEGIN TRAN;

UPDATE Production.Products
    SET unitprice += 1.00
    WHERE productid = 2;
```

Перейдите к **Соединению 2**, запустите новую транзакцию, обновите строку с товаром под номером 2 в таблице **Sales.OrderDetails** и оставьте транзакцию открытой.

```
BEGIN TRAN;

UPDATE Sales.OrderDetails
    SET unitprice += 1.00
    WHERE productid = 2;
```

На данном этапе **Соединение 1** удерживает монопольную блокировку для строки в таблице **Production.Products**, а **Соединение 2** делает то же самое в контексте таблицы **Sales.OrderDetails**. Оба запроса завершились удачно, и ни одна транзакция не была заблокирована.

Теперь, находясь внутри **Соединения 1**, попытайтесь запросить из таблицы **Sales.OrderDetails** строку с товаром под номером 2, после чего подтвердите транзакцию (не забудьте раскрыть комментарий, если вы работаете с SQL Database).

```
SELECT orderid, productid, unitprice
FROM Sales.OrderDetails -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;

COMMIT TRAN;
```

Чтобы выполнить операцию чтения, транзакция в **Соединении 1** должна получить разделяемую блокировку. Но так как этот ресурс уже монопольно заблокирован другой транзакцией, **Соединение 1** останавливает свою работу. На данном этапе происходит блокирование (пока что не взаимное). Хотя все еще остается вероятность того, что **Соединение 2** завершит свою транзакцию, освободит все блокировки и позволит **Соединению 1** получить доступ к ресурсу.

Теперь, находясь в контексте **Соединения 2**, попытайтесь запросить из таблицы **Product.Production** строку с товаром под номером 2 и затем подтвердите транзакцию.

```
SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;

COMMIT TRAN;
```

Чтобы выполнить операцию чтения, транзакция в **Соединении 2** должна получить разделяемую блокировку для строки с товаром под номером 2, которая находится в таблице **Product.Production**. Это действие конфликтует с монопольной блокировкой, удерживаемой **Соединением 1**. Процессы блокируют друг друга — то есть мы получили взаимное блокирование. Чтобы решить эту проблему, **SQL Server** выбирает один из двух процессов (как правило, в течение нескольких секунд) и прерывает его транзакцию, возвращая следующую ошибку.

Сообщение 1205, уровень 13, состояние 51, строка 1  
Транзакция (идентификатор процесса 52) вызвала взаимоблокировку  
ресурсов блокировка с другим процессом и стала жертвой  
взаимоблокировки. Запустите транзакцию повторно.

В этом примере **SQL Server** прервал транзакцию в **Соединении 1** (с идентификатором 52). Но то же самое могло быть и с **Соединением 2**, поскольку мы не устанавливали приоритетов и обе транзакции выполнили одинаковый объем работы.

Взаимное блокирование — довольно ресурсоемкое явление, так как оно требует отмены той работы, которая уже была проделана. Чтобы минимизировать возможность возникновения подобных ситуаций в своей системе, возьмите на вооружение несколько советов.

Очевидно, что чем больше длится транзакция, тем дольше удерживается блокировка и тем выше вероятность взаимного блокирования. Вам следует делать транзакции как можно более короткими, вынося за их рамки те операции, которые с логической точки зрения не являются частью единого целого.

Взаимное блокирование происходит в том случае, когда транзакции обращаются к одним и тем же ресурсам в разном порядке. В вышеприведенном примере **Соединение 1** сначала запрашивало таблицу **Production.Products**, а потом — **Sales.OrderDetails**, тогда как **Соединение 2** делало все наоборот. Этого бы не произошло, если бы обе транзакции действовали в одном порядке. Вы можете решить данную проблему, поменяв местами вышеупомянутые таблицы в одном из соединений (конечно, если это не повлияет на логику работы приложения).

В данном примере можно наблюдать настоящий логический конфликт, так как обе стороны пытаются получить доступ к одним и тем же строкам. Взаимное блокирование может случиться и без этого, например из-за отсутствия подходящих индексов для поддержки фильтров в запросе. Представьте, что обе команды во втором соединении пытаются получить товар под номером 5. Таким образом, транзакции обращаются к разным ресурсам, поэтому теоретически конфликтов быть не должно. Если в таблицах не содержатся индексы для столбца **productid**, ядру БД придется сканировать (и, следовательно, блокировать) все имеющиеся строки, чтобы выполнить фильтрацию. Естественно, это может привести к взаимному блокированию. Проще говоря, если в логике запросов нет настоящего логического конфликта, грамотная структура индексов может минимизировать вероятность возникновения подобных проблем.

Еще один момент, который следует учитывать при решении проблем со взаимным блокированием, — это выбор уровня изоляции. В нашем примере команда **SELECT** нуждается в разделяемой блокировке, поскольку она работает в режиме **READ COMMITTED**. Если вы установите уровень изоляции **READ COMMITTED**

SNAPSHOT, потребность в разделяемой блокировке отпадет и, следовательно, вы избавитесь от еще одной потенциальной причины возникновения взаимного блокирования.

Закончив с примерами, откройте любое соединение и запустите следующий код, чтобы привести БД в исходное состояние.

```
UPDATE Production.Products
    SET unitprice = 19.00
WHERE productid = 2;

UPDATE Sales.OrderDetails
    SET unitprice = 19.00
WHERE productid = 2
AND orderid >= 10500;

UPDATE Sales.OrderDetails
    SET unitprice = 15.20
WHERE productid = 2
AND orderid < 10500;
```

## В заключение

В этой главе вы познакомились с различными аспектами параллелизма, узнали, что такое транзакции и как они работают в рамках SQL Server. Я объяснил, как защищать данные от несогласованного использования внешними транзакциями и каким образом можно решать проблемы с блокированием. Был описан процесс управления степенью согласованности данных посредством выбора разных уровней изоляции и то, как этот выбор влияет на параллельный доступ. Мы рассмотрели четыре режима, в которых используются блокировки, и еще два, основанных на управлении версиями строк. В конце уделили внимание взаимному блокированию и методикам, используя которые можно уменьшить вероятность возникновения этой проблемы.

Чтобы закрепить полученные знания, выполните следующие упражнения.

## Упражнения

В этом разделе мы еще раз пройдемся по темам, которые были рассмотрены в данной главе. В предыдущих главах упражнения решались путем написания запросов или команд на языке T-SQL. Здесь все совсем иначе. Вам будут даны инструкции, следуя которым, можно решить те или иные проблемы, связанные с взаимным блокированием, и рассмотреть поведение разных уровней изоляции. Поэтому отдельного раздела с решениями здесь не будет.

Чтобы все примеры работали в контексте БД **TSQL2012**, выполните следующий код.

```
USE TSQL2012;
```



**Упражнения с 1-1 по 1-6 посвящены проблемам блокирования.**

## 1-1

Откройте в SQL Server Management Studio три соединения (назовем их **Соединение 1**, **Соединение 2** и **Соединение 3**). Запустите в первом из них следующий код, чтобы обновить строки в таблице **Sales.OrderDetails**.

```
BEGIN TRAN;

UPDATE Sales.OrderDetails
    SET discount = 0.05
    WHERE orderid = 10249;
```

## 1-2

Обратитесь к таблице **Sales.OrderDetails** в контексте **Соединения 2**; вы будете заблокированы (не забудьте раскрыть комментарий, если работаете с SQL Database).

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails -- WITH (READCOMMITTEDLOCK)
WHERE orderid = 10249;
```

## 1-3

Запустите в рамках **Соединения 3** код, представленный ниже; отследите блокировки и идентификаторы процессов, вовлеченных в цепочку блокирования.

```
SELECT -- используйте *, чтобы вывести все доступные атрибуты
    request_session_id      AS spid,
    resource_type            AS restype,
    resource_database_id     AS dbid,
    resource_description     AS res,
    resource_associated_entity_id AS resid,
    request_mode             AS mode,
    request_status           AS status
FROM sys.dm_tran_locks;
```

## 1-4

Поменяйте идентификаторы процессов 52 и 53 на те, которые вы обнаружили в цепочке блокирования в предыдущем упражнении. Запустите следующий код, чтобы получить информацию о соединении, сессии и блокировках, связанных с процессом, который участвует в цепочке блокирования.

```
-- Информация о соединении:
SELECT -- use * to explore
    session_id AS spid,
    connect_time,
    last_read,
    last_write,
```

```

    most_recent_sql_handle
FROM sys.dm_exec_connections
WHERE session_id IN(52, 53);

-- Информация о сессии:
SELECT -- use * to explore
    session_id AS spid,
    login_time,
    host_name,
    program_name,
    login_name,
    nt_user_name,
    last_request_start_time,
    last_request_end_time
FROM sys.dm_exec_sessions
WHERE session_id IN(52, 53);

-- Информация о блокировках:
SELECT -- use * to explore
    session_id AS spid,
    blocking_session_id,
    command,
    sql_handle,
    database_id,
    wait_type,
    wait_time,
    wait_resource
FROM sys.dm_exec_requests
WHERE blocking_session_id > 0;

```

## 1-5

Запустите следующий код, чтобы получить исходный текст соединений (на языке SQL), вовлеченных в цепочку блокирования.

```

SELECT session_id, text
FROM sys.dm_exec_connections
    CROSS APPLY sys.dm_exec_sql_text(most_recent_sql_handle) AS ST
WHERE session_id IN(52, 53);

```

## 1-6

Запустите внутри **Соединения 1** следующий код, чтобы откатить транзакцию.

```
ROLLBACK TRAN;
```

Тем временем команда SELECT, запущенная в рамках **Соединения 2**, вернула из таблицы **OrderDetails** две строки, содержимое которых не менялось.

Закройте все соединения. Напомню, что прервать выполнение блокирующей транзакции можно прибегнув к команде KILL.

**Упражнения с 2-1 по 2-6 посвящены разным уровням изоляции.**

## 2-1

В этом упражнении вы поработаете с уровнем изоляции `READ UNCOMMITTED`.

### 2-1a

Откройте два новых соединения (назовем их **Соединение 1** и **Соединение 2**).

### 2-1б

Запустите в контексте первого соединения код, представленный ниже, чтобы обновить и затем извлечь содержимое таблицы `Sales.OrderDetails`.

```
BEGIN TRAN;

UPDATE Sales.OrderDetails
    SET discount += 0.05
WHERE orderid = 10249;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

### 2-1в

Перейдите к **Соединению 2**, установите уровень изоляции `READ UNCOMMITTED` и выполните запрос к таблице `Sales.OrderDetails`.

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Как видите, вы получили измененную, неподтвержденную версию строк.

### 2-1г

Запустите в рамках **Соединения 1** следующий код, чтобы откатить транзакцию.

```
ROLLBACK TRAN;
```

## 2-2

В этом упражнении вы поработаете с уровнем изоляции `READ COMMITTED`.

### 2-2a

Запустите в рамках **Соединения 1** код, представленный ниже, чтобы обновить и запросить содержимое таблицы `Sales.OrderDetails`.

```
BEGIN TRAN;

UPDATE Sales.OrderDetails
    SET discount += 0.05
WHERE orderid = 10249;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

## 2-2б

Запустите в контексте **Соединения 2** следующий код, который устанавливает уровень изоляции `READ COMMITTED` и выполняет запрос к таблице `Sales.OrderDetails` (не забудьте раскрыть комментарий, если вы работаете с SQL Database).

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails -- WITH (READCOMMITTEDLOCK)
WHERE orderid = 10249;
```

Обратите внимание, что ваша транзакция заблокирована.

## 2-2в

Подтвердите транзакцию внутри **Соединения 1**.

```
COMMIT TRAN;
```

## 2-2г

Перейдите к **Соединению 2**. Как видите, вы получили измененную, подтвержденную версию строки.

## 2-2д

Верните таблицу в исходное состояние с помощью следующего кода.

```
UPDATE Sales.OrderDetails
    SET discount = 0.00
WHERE orderid = 10249;
```

## 2-3

В этом упражнении вы поработаете с уровнем изоляции `REPEATABLE READ`.

## 2-3а

Запустите в контексте **Соединения 1** следующий код, который устанавливает уровень изоляции `REPEATABLE READ` и выполняет запрос к таблице `Sales.OrderDetails`.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRAN;
```

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

Вы получите две строки со значением 0,00 в столбце **discount**.

## 2-3б

Запустите в рамках **Соединения 2** код, представленный ниже. Обратите внимание, что транзакция заблокирована.

```
UPDATE Sales.OrderDetails
SET discount += 0.05
WHERE orderid = 10249;
```

## 2-3в

Теперь снова прочитайте данные в контексте **Соединения 1** и подтвердите транзакцию.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

```
COMMIT TRAN;
```

Вы опять получите две строки со значением 0,00 в столбце **discount**, что будет свидетельствовать о повторяющемся чтении. Если бы ваш код работал на более низком уровне изоляции (таком как `READ UNCOMMITTED` или `READ COMMITTED`), команда `UPDATE` не была бы заблокирована и вы бы получили неповторяющееся чтение.

## 2-3г

Перейдите к **Соединению 2**. Как видите, обновление завершено.

## 2-3д

Верните таблицу в исходное состояние с помощью следующего кода.

```
UPDATE Sales.OrderDetails
SET discount = 0.00
WHERE orderid = 10249;
```

## 2-4

В этом упражнении вы поработаете с уровнем изоляции `SERIALIZABLE`.

## 2-4а

Перейдите к **Соединению 1**, установите уровень изоляции `SERIALIZABLE` и выполните запрос к таблице `Sales.OrderDetails`.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

## 2-4б

Переключитесь на **Соединение 2**. Попробуйте добавить в таблицу **Sales.OrderDetails** строку с идентификатором заказа из предыдущего запроса. Соединение заблокируется.

```
INSERT INTO Sales.OrderDetails
    (orderid, productid, unitprice, qty, discount)
VALUES(10249, 2, 19.00, 10, 0.00);
```

Если бы ваш код работал на более низком уровне изоляции (таком как **READ UNCOMMITTED**, **READ COMMITTED** или **REPEATABLE READ**), команда **INSERT** не была бы остановлена.

## 2-4в

Запустите в контексте **Соединения 1** следующий код, который опять выполняет запрос к таблице **Sales.OrderDetails** и подтверждает транзакцию.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

```
COMMIT TRAN;
```

Результат будет таким же, как на этапе 2-4а, но поскольку команда **INSERT** была заблокирована, фантомное чтение исключено.

## 2-4г

Вернитесь к **Соединению 2**. Команда **INSERT** должна была завершиться.

## 2-4д

Верните таблицу в исходное состояние.

```
DELETE FROM Sales.OrderDetails
WHERE orderid = 10249
    AND productid = 2;
```

## 2-4е

Запустите следующий код в рамках обоих соединений, чтобы установить стандартный уровень изоляции.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

## 2-5

В этом упражнении вы поработаете с уровнем изоляции SNAPSHOT.

### 2-5а

Если вы имеете дело с локальным экземпляром SQL Server, запустите следующий код, чтобы установить для БД TSQL2012 уровень изоляции SNAPSHOT (в SQL Database он включен по умолчанию).

### 2-5б

Запустите внутри **Соединения 1** следующий код, который открывает транзакцию, обновляет строки таблицы **Sales.OrderDetails** и возвращает их в качестве результата.

```
BEGIN TRAN;

UPDATE Sales.OrderDetails
    SET discount += 0.05
WHERE orderid = 10249;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

### 2-5в

Перейдите к **Соединению 2**, установите уровень изоляции SNAPSHOT и выполните запрос к таблице **Sales.OrderDetails**. Как видите, вы не заблокированы, вместо этого вам нужно получить старую, подтвержденную версию данных, которая была доступна на момент открытия транзакции (со скидкой, равной 0,00).

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;

BEGIN TRAN;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

### 2-5г

Вернитесь к **Соединению 1** и подтвердите транзакцию.

```
COMMIT TRAN;
```

### 2-5д

Опять запросите данные в контексте **Соединения 2**; обратите внимание, что скидка по-прежнему равна 0,00.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

## 2-5е

Находясь в том же соединении, подтвердите транзакцию и снова запросите данные; как видите, вы получили скидку размером 0,05.

```
COMMIT TRAN;
```

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

## 2-5ж

Верните таблицу к исходному состоянию.

```
UPDATE Sales.OrderDetails
    SET discount = 0.00
WHERE orderid = 10249;
```

Закройте все соединения.

## 2-6

В этом упражнении вы поработаете с уровнем изоляции READ COMMITTED SNAPSHOT.

### 2-6а

Если вы имеете дело с локальным экземпляром SQL Server, активируйте для БД TSQ2012 параметр READ\_COMMITTED\_SNAPSHOT (в SQL Database он включен по умолчанию).

```
ALTER DATABASE TSQ2012 SET READ_COMMITTED_SNAPSHOT ON;
```

### 2-6б

Откройте два новых соединения (назовем их **Соединение 1** и **Соединение 2**).

### 2-6в

Запустите в контексте **Соединения 1** следующий код, который открывает транзакцию, обновляет строки в таблице Sales.OrderDetails и возвращает их в качестве результата.

```
BEGIN TRAN;
```

```
UPDATE Sales.OrderDetails
    SET discount += 0.05
WHERE orderid = 10249;
```

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```



## 2-6г

Попробуйте запустить код, представленный ниже, внутри **Соединения 2**, которое благодаря включенному параметру `READ_COMMITTED_SNAPSHOT` работает в режиме `READ COMMITTED SNAPSHOT`. Обратите внимание, что сейчас вы не заблокированы. Вместо этого вам необходимо получить старую, подтвержденную версию данных, которая была доступна на момент запуска команды (со скидкой, равной 0,00).

```
BEGIN TRAN;

SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;
```

## 2-6д

Перейдите к **Соединению 1** и подтвердите транзакцию.

```
COMMIT TRAN;
```

## 2-6е

Вернитесь к **Соединению 2**; снова запросите данные и подтвердите транзакцию. Обратите внимание, что на этот раз скидка равна 0,05.

```
SELECT orderid, productid, unitprice, qty, discount
FROM Sales.OrderDetails
WHERE orderid = 10249;

COMMIT TRAN;
```

## 2-6ж

Верните таблицу к исходному состоянию.

```
UPDATE Sales.OrderDetails
SET discount = 0.00
WHERE orderid = 10249;
```

Закройте все соединения.

## 2-6з

Если вы имеете дело с локальным экземпляром SQL Server, присвойте параметрам БД значения по умолчанию, чтобы отключить уровни изоляции, основанные на управлении версиями строк.

```
ALTER DATABASE TSQL2012 SET ALLOW_SNAPSHOT_ISOLATION OFF;
ALTER DATABASE TSQL2012 SET READ_COMMITTED_SNAPSHOT OFF;
```

**Следующее упражнение (шаги с 1 по 7) посвящено взаимному блокированию.**

### 3-1

Откройте два новых соединения (назовем их **Соединение 1** и **Соединение 2**).

### 3-2

Запустите в контексте **Соединения 1** следующий код, который открывает транзакцию и обновляет в таблице **Production.Products** строку с товаром под номером 2.

```
BEGIN TRAN;

UPDATE Production.Products
    SET unitprice += 1.00
WHERE productid = 2;
```

### 3-3

Запустите в контексте **Соединения 2** следующий код, который открывает транзакцию и обновляет в таблице **Production.Products** строку с товаром под номером 3.

```
BEGIN TRAN;

UPDATE Production.Products
    SET unitprice += 1.00
WHERE productid = 3;
```

### 3-4

Запросите продукт под номером 3 в **Соединении 1**. Вы будете заблокированы (не забудьте раскрыть комментарий с табличным указанием, если вы работаете с SQL Database).

```
SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 3;

COMMIT TRAN;
```

### 3-5

Запросите продукт под номером 2 во втором соединении. Выполнение вашего запроса приостановится, а в одном из соединений будет сгенерировано сообщение о взаимном блокировании.

```
SELECT productid, unitprice
FROM Production.Products -- WITH (READCOMMITTEDLOCK)
WHERE productid = 2;

COMMIT TRAN;
```

### 3-6

Можете ли вы предложить решение, которое позволило бы избежать этой проблемы? Подсказки ищите в разделе «Взаимное блокирование».

### 3-7

Приведите таблицу **Products** к исходному состоянию.

```
UPDATE Production.Products  
    SET unitprice = 19.00  
WHERE productid = 2;
```

```
UPDATE Production.Products  
    SET unitprice = 10.00  
WHERE productid = 3;
```

# Глава 10

## ПРОГРАММИРУЕМЫЕ ОБЪЕКТЫ

В данной главе мы кратко рассмотрим концепцию программируемых объектов и познакомимся с возможностями, которые Microsoft SQL Server предоставляет в этой области. Вы узнаете, что такое переменные, пакеты, инструкции для управления потоком выполнения, курсоры, временные таблицы, процедуры (пользовательские функции, триггеры и хранимые процедуры) и динамические элементы языка SQL. Мы не станем углубляться в детали и технические тонкости тех или иных языковых конструкций; в приоритете будут логические аспекты и возможности программируемых объектов.

### Переменные

**Переменные** предназначены для временного хранения данных и их дальнейшего использования в рамках одного и того же пакета. К пакетам мы вернемся позже; сейчас вам достаточно знать, что это одна или несколько команд, которые передаются в SQL Server и выполняются как единое целое.

Для объявления переменных используется команда `DECLARE`; с помощью команды `SET` происходит присваивание значений. Например, следующий код объявляет переменную под названием `@i` типа `INT` и присваивает ей число 10.

```
DECLARE @i AS INT;  
SET @i = 10;
```

SQL Server 2008 и SQL Server 2012 поддерживают объявление и инициализацию переменных в рамках одной команды, как показано ниже.

```
DECLARE @i AS INT = 10;
```

Значение, которое присваивается скалярной переменной, должно быть результатом скалярного выражения (даже если речь идет о вложенном запросе). Например, следующий код объявляет переменную под названием `@empname` и присваивает ей результат выполнения вложенного скалярного запроса, который возвращает полное имя сотрудника с идентификатором 3.

```
USE TSQL2012;  
  
DECLARE @empname AS NVARCHAR(31);  
  
SET @empname = (SELECT firstname + N' ' + lastname  
                FROM HR.Employees  
                WHERE empid = 3);  
  
SELECT @empname AS empname;
```

Результирующий набор будет выглядеть так.

```
empname
-----
Джуди  Лью
```

Команда SET может одновременно работать только с одной переменной, поэтому чтобы присвоить значения нескольким атрибутам, вам понадобится соответствующее количество таких команд. В связи с этим получение содержимого нескольких атрибутов одной и той же строки может привести к избыточности вашего кода. В примере, представленном ниже, используются две отдельные команды SET, которые присваивают переменным имя и фамилию сотрудника с идентификатором 3.

```
DECLARE @firstname AS NVARCHAR(10), @lastname AS NVARCHAR(20);

SET @firstname = (SELECT firstname
                  FROM HR.Employees
                  WHERE empid = 3);
SET @lastname = (SELECT lastname
                 FROM HR.Employees
                 WHERE empid = 3);

SELECT @firstname AS firstname, @lastname AS lastname;
```

Этот код возвращает следующий результат.

```
firstname  lastname
-----
Джуди      Лью
```

SQL Server также поддерживает нестандартную версию инструкции SELECT, которая позволяет присваивать переменным множественные значения, полученные из одной строки. Пример показан ниже.

```
DECLARE @firstname AS NVARCHAR(10), @lastname AS NVARCHAR(20);

SELECT
    @firstname = firstname,
    @lastname = lastname
FROM HR.Employees
WHERE empid = 3;

SELECT @firstname AS firstname, @lastname AS lastname;
```

Этот синтаксис имеет предсказуемое поведение, когда возвращается всего одна строка. Даже если строк несколько, код не завершается ошибкой; в данном случае содержимое переменных перезаписывается при доступе к каждой итоговой записи. Таким образом, переменные будут содержать значения той строки, которая оказалась последней в результирующем наборе. Например, следующая команда SELECT находит две подходящие строки.

```
DECLARE @empname AS NVARCHAR(31);

SELECT @empname = firstname + N' ' + lastname
FROM HR.Employees
```

```
WHERE mgrid = 2;

SELECT @empname AS empname;
```

Информация о сотруднике, которая оказывается внутри переменной после выполнения инструкции `SELECT`, зависит от того, в каком порядке `SQL Server` перебирает полученные строки, причем этот порядок невозможно изменить. В моем случае результат получился следующим.

```
empname
-----
Свен Бак
```

Инструкция `SET` более безопасна, чем `SELECT`, поскольку для извлечения данных из таблицы она использует вложенные скалярные запросы. Как вы помните, скалярный запрос, возвращающий больше одного значения, завершается ошибкой. Например, следующий код не может быть выполнен успешно.

```
DECLARE @empname AS NVARCHAR(31);

SET @empname = (SELECT firstname + N' ' + lastname
                FROM HR.Employees
                WHERE mgrid = 2);

SELECT @empname AS empname;
```

Поскольку переменная не была инициализирована, она по-прежнему хранит отметку `NULL` (значение по умолчанию). Результат выполнения этого кода представлен ниже.

```
Сообщение 512, уровень 16, состояние 1, строка 3
Вложенный запрос вернул больше одного значения. Это запрещено, когда
вложенный запрос следует после =, !=, <, <=, >, >= или используется
в качестве выражения.
empname
-----
NULL
```

## Пакеты

**Пакет** — это одна или несколько команд языка `T-SQL`, отправленных клиентским приложением в `SQL Server` для их дальнейшего выполнения в виде единого целого. К пакету применяются такие этапы обработки, как синтаксический анализ, разрешение имен объектов и столбцов, проверка полномочий и оптимизация.

Не стоит путать пакеты с транзакциями. Последние представляют собой атомарную единицу работы. Один пакет может содержать несколько транзакций, а одна транзакция может быть передана в виде нескольких пакетов. Если транзакция отменяется во время выполнения, `SQL Server` откатывает затронутые ею данные в исходное состояние (в котором они пребывали на момент открытия транзакции), вне зависимости от того, где начинался пакет.

Клиентские программные интерфейсы, такие как ADO.NET, предоставляют средства для отправки пакетов на выполнение в SQL Server. Утилиты SQL Server Management Studio, SQLCMD и OSQL поддерживают команду под названием GO, которая обозначает конец пакета. Стоит отметить, что это клиентская команда, то есть она выполняется не на сервере.

## Анализ

Пакет — это набор команд, которые анализируются и выполняются как единое целое. Переход к этапу выполнения происходит только после успешного анализа. В случае обнаружения синтаксической ошибки отправка пакета в SQL Server отменяется. Например, следующий код состоит из трех пакетов, причем во втором из них есть опечатка (FOM вместо FROM во втором запросе).

```
-- Корректный пакет
PRINT 'Первый пакет';
USE TSQL2012;
GO
-- Некорректный пакет
PRINT 'Второй пакет';
SELECT custid FROM Sales.Customers;
SELECT orderid FOM Sales.Orders;
GO
-- Корректный пакет
PRINT 'Третий пакет';
SELECT empid FROM HR.Employees;
```

Так как второй пакет содержит синтаксическую ошибку, он не будет отправлен на выполнение. Первый и третий пакеты будут выполнены, поскольку они успешно проходят проверку. В этом можно убедиться, взглянув на результат выполнения кода.

```
Неправильный синтаксис около конструкции "Sales".
Третий пакет
empid
-----
2
7
1
5
6
8
3
9
4

(строк обработано: 9)
```

## Пакеты и переменные

Область видимости переменной ограничивается пакетом, в котором она объявлена. При попытке сослаться на нее из другого пакета вы получите сообщение об ошибке, в котором будет сказано, что переменная не была определена. Напри-

мер, следующий код пытается вывести содержимое переменной в контексте двух пакетов, в одном из которых она была объявлена.

```
DECLARE @i AS INT;
SET @i = 10;
-- Работает
PRINT @i;
GO

-- Завершается ошибкой
PRINT @i;
```

Ссылка на переменную, которая указана в первой команде PRINT, является корректной, поскольку она находится в «родном» для переменной пакете. Со второй командой PRINT все с точностью наоборот. Таким образом, сначала мы получим значение переменной (10), а затем сообщение об ошибке.

```
10
Сообщение 137, уровень 15, состояние 2, строка 2
Необходимо объявить скалярную переменную "@i".
```

## Команды, которые не могут находиться в одном пакете с другими командами

Следующие команды нельзя использовать в одном пакете с любыми другими: CREATE DEFAULT, CREATE FUNCTION, CREATE PROCEDURE, CREATE RULE, CREATE SCHEMA, CREATE TRIGGER и CREATE VIEW. Например, код, представленный ниже, является некорректным, поскольку в нем в рамках одного пакета находятся команды IF и CREATE VIEW.

```
IF OBJECT_ID('Sales.MyView', 'V') IS NOT NULL DROP VIEW Sales.MyView;

CREATE VIEW Sales.MyView
AS

SELECT YEAR(orderdate) AS orderyear, COUNT(*) AS numorders
FROM Sales.Orders
GROUP BY YEAR(orderdate);
GO
```

Попытка запустить этот код приведет к следующей ошибке.

```
Сообщение 111, уровень 15, состояние 1, строка 2
CREATE VIEW должна быть первой инструкцией в пакетном запросе.
```

Чтобы обойти эту проблему, разнесите команды по отдельным пакетам, указав GO после первой из них.

## Разрешение имен

Пакет можно считать единицей разрешения имен, что означает следующее: проверка существования объектов и столбцов происходит на уровне пакета. Помните об этом, когда будете обозначать границы своих пакетов с помощью команды GO.



Например, изменение структуры объекта и его содержимого в рамках одного пакета может привести к ошибке разрешения имен, поскольку SQL Server будет руководствоваться старой структурой объекта. Я попытаюсь продемонстрировать эту проблему на реальном примере и дам совет, как ее решить.

Запустите следующий код, чтобы создать внутри текущей БД таблицу T1 с единственным столбцом col1.

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;  
CREATE TABLE dbo.T1(col1 INT);
```

Теперь добавьте столбец col2 и обратитесь к нему в рамках того самого пакета.

```
ALTER TABLE dbo.T1 ADD col2 INT;  
SELECT col1, col2 FROM dbo.T1;
```

Несмотря на то что код выглядит абсолютно корректным, его выполнение будет прервано на этапе разрешения имен. Вы получите следующую ошибку.

```
Сообщение 207, уровень 16, состояние 1, строка 2  
Недопустимое имя столбца "col2".
```

В момент, когда внутри команды SELECT происходило разрешение имен, таблица T1 имела всего один столбец, поэтому попытка доступа к столбцу col2 привела к ошибке. Чтобы избежать подобных проблем, элементы языков DML и DDL следует выносить в разные пакеты, как показано ниже.

```
ALTER TABLE dbo.T1 ADD col2 INT;  
GO  
SELECT col1, col2 FROM dbo.T1;
```

## Параметр команды GO

Команда GO на самом деле не является частью языка T-SQL; она используется в клиентских инструментах, таких как SSMS, для обозначения конца пакета. Данная команда поддерживает параметр, который указывает, сколько раз пакет будет выполнен на сервере. Чтобы увидеть, как это работает, создайте таблицу T1, воспользовавшись следующим кодом.

```
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;  
CREATE TABLE dbo.T1(col1 INT IDENTITY);
```

Столбец col1 получает значение автоматически, благодаря свойству identity. Но мы точно так же могли бы оставить стандартные ограничения, которые подразумевают использование объектов последовательности. Теперь запустите код, представленный ниже, чтобы команды языка DML не выводили сообщение о количестве обработанных строк.

```
SET NOCOUNT ON;
```

В завершение определите границы пакета с помощью команды INSERT DEFAULT VALUES и выполните его 100 раз подряд.

```
INSERT INTO dbo.T1 DEFAULT VALUES;  
GO 100
```

```
SELECT * FROM dbo.T1;
```

Этот запрос вернет 100 строк со значениями от 1 до 100.

## Управление потоком выполнения

Язык T-SQL позволяет управлять потоком выполнения кода с помощью простых инструкций, таких как IF...ELSE и WHILE.

### Инструкция IF...ELSE

Инструкция IF...ELSE позволяет управлять потоком выполнения кода на основе предиката. Команда, которую вы укажете, будет выполняться, если предикат вернет TRUE; можно указать вторую команду на случай возврата значений FALSE или UNKNOWN.

Код, представленный ниже, позволяет узнать, является ли сегодня последним днем в году (отличается ли текущий год от того, что будет завтра). В зависимости от ответа выводится подходящее сообщение.

```
IF YEAR(SYSDATETIME()) <> YEAR(DATEADD(day, 1, SYSDATETIME()))  
    PRINT 'Сегодня последний день в году.';  
ELSE  
    PRINT 'Сегодня не последний день в году.';
```

Чтобы показать, какие части кода выполняются, а какие нет, я воспользовался командой PRINT, но вы можете использовать любые другие команды.

Не забывайте, что в языке T-SQL используется троичное исчисление, и, чтобы активировать блок ELSE, предикат может вернуть как FALSE, так и UNKNOWN. Если вам нужно по-разному обрабатывать эти значения, достаточно сделать отдельную проверку на отметки NULL, используя предикат IS NULL.

Вы можете делать потоки выполнения вложенными, если их становится больше двух. Например, код, представленный ниже, по-разному обрабатывает три возможные ситуации:

- сегодня последний день в году;
- сегодня последний день месяца, но не последний день в году;
- сегодня не последний день месяца.

```
IF YEAR(SYSDATETIME()) <> YEAR(DATEADD(day, 1, SYSDATETIME()))  
    PRINT 'Сегодня последний день в году.';  
ELSE  
    IF MONTH(SYSDATETIME()) <> MONTH(DATEADD(day, 1, SYSDATETIME()))  
        PRINT 'Сегодня последний день месяца, но не последний день в году.';  
    ELSE  
        PRINT 'Сегодня не последний день месяца.';
```

Если в рамках инструкций IF или ELSE нужно выполнить сразу несколько команд, следует использовать специальные блоки, границы которых обозначаются ключевыми словами BEGIN и END. В следующем примере показано, как запускать разные процессы в зависимости от того, является ли сегодня первым днем месяца.

```
IF DAY(SYSDATETIME()) = 1
BEGIN
    PRINT 'Сегодня первый день месяца.';
    PRINT 'Запускаем процесс первый-день-месяца.';
    /* ... здесь должен быть код процесса ... */
    PRINT 'Завершаем процесс первый-день-месяца.';
END
ELSE
BEGIN
    PRINT 'Today is not the first day of the month.';
    PRINT 'Запускаем процесс не-первый-день-месяца.';
    /* ... здесь должен быть код процесса ... */
    PRINT 'Завершаем процесс не-первый-день-месяца.';
END
```

## Инструкция WHILE

Язык T-SQL поддерживает инструкцию WHILE, которая делает возможным циклическое выполнение кода. Она выполняет одну и ту же команду или блок команд, пока предикат, указанный в конце, возвращает TRUE. Цикл прерывается при получении значений FALSE или UNKNOWN. В языке T-SQL нет элементов, которые позволяют выполнить код какое-то определенное количество раз, но такой подход можно легко эмулировать, используя инструкцию WHILE и одну переменную. Например, следующий код содержит цикл с десятью итерациями.

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    PRINT @i;
    SET @i = @i + 1;
END;
```

Здесь мы объявляем целочисленную переменную под названием @i, которая играет роль счетчика, и инициализируем ее значением 1. Затем начинается цикл, который работает до тех пор, пока @i не превысит число 10. При каждом проходе код в теле цикла выводит содержимое переменной и увеличивает ее значение на 1. Как видно по результату, представленному ниже, цикл выполняется десять раз.

```
1
2
3
4
5
6
7
8
9
10
```

Если в какой-то момент вам понадобится выйти из цикла и продолжить выполнение инструкций, которые находятся за пределами его тела, можете воспользоваться командой `BREAK`. Например, следующий код разрывает цикл, если `@i` равно 6.

```
DECLARE @i AS INT = 1;
WHILE @i <= 10
BEGIN
    IF @i = 6 BREAK;
    PRINT @i;
    SET @i = @i + 1;
END;
```

Как видно по результату, представленному ниже, цикл делает пять полных проходов и завершается в начале шестого.

```
1
2
3
4
5
```

Конечно, данный код нельзя назвать рациональным; если вы хотите, чтобы цикл выполнил пять проходов, вам просто следует указать предикат `@i <= 5`. Здесь я всего лишь хотел показать простой пример использования команды `BREAK`.

Чтобы пропустить оставшийся код в текущей итерации и вернуться к предикату цикла, используйте команду `CONTINUE`. Например, в следующем коде на шестой итерации пропускается выполнение участка кода, начиная с инструкции `IF` и заканчивая границей тела цикла.

```
DECLARE @i AS INT = 0;
WHILE @i < 10
BEGIN
    SET @i = @i + 1;
    IF @i = 6 CONTINUE;
    PRINT @i;
END;
```

Результат выполнения этого кода свидетельствует о том, что значение переменной `@i` было выведено во всех итерациях, кроме шестой.

```
1
2
3
4
5
7
8
9
10
```

## Пример использования инструкций IF и WHILE

В этом разделе демонстрируется пример совместного использования инструкций IF и WHILE. Наша задача — создать таблицу под названием `dbo.Numbers` и добавить в ее столбец `n` 1000 строк со значениями от 1 до 1000.

```
SET NOCOUNT ON;
IF OBJECT_ID('dbo.Numbers', 'U') IS NOT NULL DROP TABLE dbo.Numbers;
CREATE TABLE dbo.Numbers(n INT NOT NULL PRIMARY KEY);
GO

DECLARE @i AS INT = 1;
WHILE @i <= 1000
BEGIN
    INSERT INTO dbo.Numbers(n) VALUES(@i);
    SET @i = @i + 1;
END
```

Инструкция IF проверяет, существует ли таблица `Numbers` в текущей базе данных, и в случае положительного ответа удаляет ее. Затем цикл WHILE заполняет эту таблицу значениями от 1 до 1000, используя тысячу итераций.

## Курсоры

В главе 2 я уже объяснял, что запросы, не содержащие инструкции `ORDER BY`, возвращают множество (или мультимножество), тогда как результатом упорядоченных запросов является то, что в языке SQL называется **курсором** — нереляционный набор с четко определенным порядком следования строк. В этом контексте термин «курсор» следует считать абстрактным, однако язык T-SQL поддерживает конкретную реализацию курсоров, которая позволяет обрабатывать итоговые строки последовательно и в заданном порядке. Запросы, с которыми мы имели дело ранее, возвращали множества или мультимножества, не гарантируя упорядоченности их содержимого.

Хочу подчеркнуть, что в целом приоритет должен отдаваться реляционным запросам; курсоры следует использовать только в тех случаях, когда для этого есть веская причина. Данный совет имеет под собой несколько оснований.

1. Прежде всего, используя курсор, вы идете против реляционной модели, которая основана на теории множеств.
2. Последовательный перебор каждой записи довольно избыточен, если сравнивать с запросами, основанными на множествах. Хотя с точки зрения физической обработки эти два подхода имеют много общего, код с использованием курсоров обычно работает в несколько раз медленней.
3. Курсоры заставляют тратить много кода на физические аспекты запроса, то есть на описание механизма обработки данных (объявление и открытие курсора, перебор его записей, закрытие и освобождение). Решения, основанные на множествах, позволяют сосредоточиться на логике работы кода; главный вопрос — не как, а что нужно получить. Таким образом, запросы, использующие курсоры, получаются более длинными и запутанными, что усложняет их обслуживание.

Большинству людей, которые начинают изучать язык SQL, довольно сложно свыкнуться с концепцией множеств. В то же время курсоры выглядят вполне логично — записи обрабатываются последовательно и в определенном порядке. Этим и объясняется их достаточно широкое (и часто неоправданное) применение. Они используются там, где лучше было бы прибегнуть к реляционным запросам. Вы должны заставить себя мыслить в рамках парадигмы, основанной на множествах. Это может занять некоторое время (иногда даже годы), но в языке, который полностью опирается на реляционную модель, такой подход себя оправдывает.

Работа с курсорами похожа на рыбалку с удочкой: за раз можно выловить всего одну рыбу. Работа с множествами больше смахивает на промышленный вылов. Давайте рассмотрим еще один пример. Представьте, что у вас есть две фабрики по упаковке апельсинов: одна старая, а вторая современная. Упаковки бывают маленькими, средними и большими. Старая фабрика работает в режиме курсора: апельсины перемещаются по конвейерной ленте, на конце которой находится человек, занимающийся проверкой и расфасовкой каждого плода. Естественно, такой процесс не может быть быстрым. Кроме того, имеет значение порядок следования апельсинов — если они изначально сортируются по размеру, их распределение будет занимать значительно меньше времени. Современная фабрика работает по принципу множеств: все апельсины помещаются в большой контейнер, на дне которого имеются отверстия маленького размера. После небольшой встряски самые маленькие плоды падают вниз, а то, что осталось, высыпается в контейнер с отверстиями среднего размера, и процесс повторяется. В конце у нас остаются только крупные апельсины.

Итак, развеяв все сомнения относительно того, какой тип запросов является более предпочтительным, я позволю себе остановиться на нескольких исключениях — ситуациях, в которых действительно следует использовать курсоры. Представьте, что вам нужно выполнить некое действие с каждой строкой таблицы или представления. Это может быть, например, административная задача в рамках вашей БД. В данном случае курсор отлично подойдет для последовательной обработки элементов (будь то индекс или список имен таблиц).

Иногда случается, что запросы, основанные на множествах, демонстрируют низкую производительность, несмотря на все ваши попытки их ускорить. Выше я говорил, что реляционный подход имеет значительное преимущество в скорости, но бывают ситуации, когда все происходит наоборот. Некоторые вычисления требуют значительно менее частого обращения к таблицам, если исходные данные заранее упорядочены и обрабатываются последовательно. В качестве примера можно привести вычисление текущих агрегатов в версиях языка SQL, предшествовавших выходу SQL Server 2012. В главе 7 вы познакомились с крайне эффективным решением, основанным на оконных агрегатных функциях. Однако в старых версиях SQL Server реляционные запросы были не очень хорошо приспособлены к получению текущих агрегатов, требуя многократного извлечения одних и тех же данных. Я не стану подробно останавливаться на процессе оптимизации такого кода; достаточно сказать, что курсоры справляются с этой задачей намного лучше (по крайней мере, в SQL Server 2008 и более ранних версиях), поскольку извлекают данные всего один раз.

В начале этой главы я предупреждал, что здесь рассматриваются только самые основные моменты, но мне кажется, что пример использования курсоров не будет лишним.

В целом работу с курсором можно разделить на следующие этапы.

- Объявление курсора на основе запроса.
- Открытие курсора.
- Извлечение и сохранение внутри переменных содержимого атрибутов первой строки.
- Перебор содержимого курсора в цикле, пока не будет достигнута последняя запись (то есть функция @@FETCH\_STATUS не вернет 0); на каждой итерации производится необходимая обработка, а значения атрибутов текущей записи присваивается переменным.
- Закрытие курсора.
- Освобождение курсора.

Код, представленный ниже, вычисляет общее текущее количество товара для каждого клиента и добавляет к результату месяц, взятый из представления **Sales.CustOrders**.

```
SET NOCOUNT ON;

DECLARE @Result TABLE
(
    custid      INT,
    ordermonth  DATETIME,
    qty         INT,
    runqty      INT,
    PRIMARY KEY(custid, ordermonth)
);

DECLARE
    @custid AS INT,
    @prvcustid AS INT,
    @ordermonth DATETIME,
    @qty AS INT,
    @runqty AS INT;

DECLARE C CURSOR FAST_FORWARD /* только прямое чтение */ FOR
    SELECT custid, ordermonth, qty
    FROM Sales.CustOrders
    ORDER BY custid, ordermonth;

OPEN C;

FETCH NEXT FROM C INTO @custid, @ordermonth, @qty;

SELECT @prvcustid = @custid, @runqty = 0;

WHILE @@FETCH_STATUS = 0
    BEGIN
        IF @custid <> @prvcustid
```

```
SELECT @prvcustid = @custid, @runqty = 0;

SET @runqty = @runqty + @qty;

INSERT INTO @Result VALUES(@custid, @ordermonth, @qty, @runqty);

FETCH NEXT FROM C INTO @custid, @ordermonth, @qty;
END

CLOSE C;

DEALLOCATE C;

SELECT
    custid,
    CONVERT(VARCHAR(7), ordermonth, 121) AS ordermonth,
    qty,
    runqty
FROM @Result
ORDER BY custid, ordermonth;
```

Здесь объявляется курсор, основанный на запросе, который возвращает содержимое таблицы **CustOrders**; строки, упорядоченные по идентификатору клиента и месяцу размещения заказа, обрабатываются одна за другой. При каждой итерации общее текущее количество товара сохраняется в переменной **@runqty**, которая сбрасывается для каждого следующего клиента. Чтобы получить общее текущее количество товара для отдельной строки, мы добавляем к переменной **@runqty** месячный показатель (**@qty**) и затем вставляем полученное значение (вместе с идентификатором клиента, месяцем размещения заказа и содержимым **@qty**) в табличную переменную под названием **@Result**. После обработки всех записей, хранящихся внутри курсора, будет выполнен запрос, который выведет все, что было добавлено в табличную переменную (текущие агрегаты).

Результат (точнее, его сокращенная версия) представлен ниже.

custid	ordermonth	qty	runqty
1	2007-08	38	38
1	2007-10	41	79
1	2008-01	17	96
1	2008-03	18	114
1	2008-04	60	174
2	2006-09	6	6
2	2007-08	18	24
2	2007-11	10	34
2	2008-03	29	63
3	2006-11	24	24
3	2007-04	30	54
3	2007-05	80	134
3	2007-06	83	217
3	2007-09	102	319
3	2008-01	40	359
...			
89	2006-07	80	80
89	2006-11	105	185
89	2007-03	142	327



89	2007-04	59	386
89	2007-07	59	445
89	2007-10	164	609
89	2007-11	94	703
89	2008-01	140	843
89	2008-02	50	893
89	2008-04	90	983
89	2008-05	80	1063
90	2007-07	5	5
90	2007-09	15	20
90	2007-10	34	54
90	2008-02	82	136
90	2008-04	12	148
91	2006-12	45	45
91	2007-07	31	76
91	2007-12	28	104
91	2008-02	20	124
91	2008-04	81	205

(строк обработано: 636)

Как я уже объяснял в главе 7, SQL Server 2012 поддерживает улучшенные оконные функции, которые позволяют вычислять текущие агрегаты быстро и изящно, освобождая вас от необходимости использовать курсоры. Наш предыдущий пример можно переписать следующим образом.

```
SELECT custid, ordermonth, qty,
       SUM(qty) OVER(PARTITION BY custid
                     ORDER BY ordermonth
                     ROWS UNBOUNDED PRECEDING) AS runqty
FROM Sales.CustOrders
ORDER BY custid, ordermonth;
```

## Временные таблицы

Обычные таблицы не всегда подходят для хранения временных данных. Представьте, что у вас есть информация, которая должна быть доступна только для текущей сессии (или даже для текущего пакета). Это могут быть данные, которые нужны во время вычислительных операций, как в предыдущем примере с курсором.

Специально для таких случаев в SQL Server предусмотрено три вида временных таблиц: локальные, глобальные и табличные переменные. В следующих разделах вы узнаете, что они собой представляют, и познакомитесь с реальными примерами их применения.

### Локальные таблицы

Имена локальных временных таблиц должны начинаться со знака решетки, например #T1. Их содержимое (как и содержимое остальных двух видов временных таблиц) хранится в БД tempdb.

Локальные временные таблицы видны только в контексте сессии, в которой они были созданы; доступ к ним можно получить на том уровне, где находится их

определение, и далее по стеку вызовов (в том числе внутри вложенных процедур, функций, триггеров и динамических пакетов). Они автоматически уничтожаются, когда текущий уровень выходит за рамки пространства имен. Представьте, что у нас есть цепочка вызовов хранимых процедур, начиная с Proc1 и заканчивая Proc4. В процедуре Proc2 перед вызовом Proc3 создается локальная временная таблица под названием #T1. Она будет доступна процедурам Proc2, Proc3 и Proc4, но не Proc1; SQL Server удалит ее сразу же, как только процедура Proc2 завершит свою работу. Таблица, которая создается в произвольном пакете на самом верхнем уровне вложенности (то есть когда функция @@NESTLEVEL возвращает 0), доступна для всех последующих пакетов; ее удаление будет выполнено только после отключения сессии, в которой она была создана.

Вам, наверное, интересно, каким образом SQL Server предотвращает конфликт имен, когда две сессии создают локальные временные таблицы с одинаковыми названиями. Все очень просто: к имени каждой таблицы автоматически добавляется суффикс, который обеспечивает ее уникальность в рамках БД tempdb. Вы как разработчик можете об этом не беспокоиться: доступ к таблицам в рамках сессии осуществляется без использования внутренних суффиксов.

Одним из очевидных способов применения временных таблиц является хранение промежуточных результатов (например, во время выполнения цикла) с возможностью последующего доступа к ним.

Иногда возникает необходимость многократного обращения к данным, которые представляют собой результаты ресурсоемких вычислений. Допустим, вам нужно соединить таблицы Sales.Orders и Sales.OrderDetails, вычислить, какое количество товара было заказано в каждом году, затем попарно объединить полученные результаты для сравнения годовых показателей. В нашей БД таблицы Orders и OrderDetails имеют очень маленький объем, но в реальных условиях количество записей может измеряться миллионами. Мы могли бы воспользоваться табличными выражениями, но, как вы помните, они являются сугубо виртуальными объектами; следовательно, вся работа по сканированию данных, соединению таблиц и агрегированию будет выполнена два раза. Чтобы не повторять все эти ресурсоемкие операции, результаты проще сохранить в локальной временной таблице; после этого достаточно взять два экземпляра имеющихся у нас данных и выполнить соединение. В нашем случае это будет крайне эффективно, поскольку результирующий набор будет содержать малое количество строк — по одной для каждого года.

Эта ситуация проиллюстрирована на примере следующего кода.

```
IF OBJECT_ID('tempdb.dbo.#MyOrderTotalsByYear') IS NOT NULL
    DROP TABLE dbo.#MyOrderTotalsByYear;
GO

CREATE TABLE #MyOrderTotalsByYear
(
    orderyear INT NOT NULL PRIMARY KEY,
    qty INT NOT NULL
);

INSERT INTO #MyOrderTotalsByYear(orderyear, qty)
SELECT
    YEAR(O.orderdate) AS orderyear,
```

```

        SUM(OD.qty) AS qty
    FROM Sales.Orders AS O
        JOIN Sales.OrderDetails AS OD
            ON OD.orderid = O.orderid
    GROUP BY YEAR(orderdate);

SELECT Cur.orderyear, Cur.qty AS curyearqty, Prv.qty AS prvyearqty
FROM dbo.#MyOrderTotalsByYear AS Cur
    LEFT OUTER JOIN dbo.#MyOrderTotalsByYear AS Prv
        ON Cur.orderyear = Prv.orderyear + 1;

```

Результат показан ниже.

orderyear	curyearqty	prvyearqty
-----	-----	-----
2006	9581	NULL
2007	25489	9581
2008	16247	25489

Чтобы убедиться в том, что локальная временная таблица доступна только в рамках сессии, в которой она была создана, попробуйте обратиться к ней извне.

```
SELECT orderyear, qty FROM dbo.#MyOrderTotalsByYear;
```

Вы получите следующую ошибку.

```
Сообщение 208, уровень 16, состояние 0, строка 1
Недопустимое имя объекта "#MyOrderTotalsByYear".
```

Когда закончите, вернитесь к исходной сессии и удалите временную таблицу.

```
IF OBJECT_ID('tempdb.dbo.#MyOrderTotalsByYear') IS NOT NULL
    DROP TABLE dbo.#MyOrderTotalsByYear;
```

В целом удаление ресурсов рекомендуется проводить сразу после завершения работы с ними.

## Глобальные таблицы

Глобальная временная таблица доступна для всех сессий и обозначается с помощью двойного знака решетки, например `##T1`. Ее удаление происходит в момент отключения сессии, в которой она была создана (при условии отсутствия внешних ссылок).



### ПРИМЕЧАНИЕ

На момент написания этой книги глобальные временные таблицы не поддерживались в Windows Azure SQL Database, поэтому примеры, приведенные в данном разделе, следует запускать с помощью локального экземпляра SQL Server.

Глобальные временные таблицы используются для хранения общедоступных промежуточных данных. Для обращения к ним (в том числе с помощью элементов DDL и DML) не требуется никаких специальных полномочий. Конечно, это означает, что любой пользователь может изменять или даже удалять их содержимое,

поэтому прежде чем использовать их в своих запросах, следует взвесить все за и против.

Следующий код создает глобальную временную таблицу под названием **##Globals**, которая содержит столбцы **id** и **val**.

```
CREATE TABLE dbo.##Globals
(
    id sysname          NOT NULL PRIMARY KEY,
    val SQL_VARIANT NOT NULL
);
```

Эта таблица выполняет те же функции, что и глобальные переменные (которые, к слову, не поддерживаются в SQL Server). Столбцы **id** и **val** имеют типы данных **SYSNAME** (используется для внутреннего представления идентификаторов) и **SQL\_VARIANT** (универсальный тип, совместимый практически с любыми данными).

Доступ к данной таблице открыт для всех. Например, следующий код добавляет в нее строку, которая представляет переменную **i**, и присваивает ей целочисленное значение 10.

```
INSERT INTO dbo.##Globals(id, val) VALUES(N'i', CAST(10 AS INT));
```

Вы можете также изменять и извлекать данные из этой таблицы. Запустите следующий код в любой сессии, чтобы получить текущее значение переменной **i**.

```
SELECT val FROM dbo.##Globals WHERE id = N'i';
```

Результат показан ниже.

```
val
-----
10
```



#### ПРИМЕЧАНИЕ

Не стоит забывать, что глобальная временная таблица автоматически уничтожается сразу после отключения сессии, в которой она была создана (при условии отсутствия внешних ссылок).

Если вы хотите, чтобы глобальная временная таблица создавалась при каждом запуске SQL Server и была доступна после завершения работы, вам необходимо использовать хранимую процедуру, помеченную параметром **startup** (подробности ищите в разделе **sp\_procoption** электронного справочника по адресу [msdn.microsoft.com/ru-ru/library/ms181720.aspx](https://msdn.microsoft.com/ru-ru/library/ms181720.aspx)).

Чтобы вручную удалить глобальную временную таблицу, откройте любую сессию и выполните следующий код.

```
DROP TABLE dbo.##Globals
```

## Табличные переменные

Табличные переменные, которые объявляются с помощью уже знакомой вам команды `DECLARE`, имеют много общего с локальными временными таблицами.

Как и другие виды временных таблиц, табличные переменные физически хранятся в БД `tempdb` (вопреки расхожему мнению о том, что они находятся исключительно в оперативной памяти). Их область видимости ограничена не только сессией, в которой они были созданы, но и текущим пакетом. При этом доступ к ним закрыт даже для вложенных и всех последующих пакетов, объявленных в рамках сессии.

При откате транзакции, которая была объявлена вручную, отменяются все изменения, внесенные ею во временные таблицы; однако в случае с табличными переменными отменяется только действие последней команды, которая завершилась ошибкой или не успела выполняться до конца.

Временные таблицы и табличные переменные отличаются в плане оптимизации, однако это уже тема для другой книги. Достаточно сказать, что табличные переменные обычно работают быстрее с минимальными объемами данных (в несколько строк); во всех остальных случаях временные таблицы показывают более высокую производительность.

Давайте перепишем наш пример, в котором мы вычисляли общее годовое количество заказанного товара, сравнивая его с аналогичным показателем за предыдущий год. На этот раз воспользуемся табличной переменной.

```
DECLARE @MyOrderTotalsByYear TABLE
(
    orderyear INT NOT NULL PRIMARY KEY,
    qty       INT NOT NULL
);

INSERT INTO @MyOrderTotalsByYear(orderyear, qty)
SELECT
    YEAR(O.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
        ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);

SELECT Cur.orderyear, Cur.qty AS curyearqty, Prv.qty AS prvyearqty
FROM @MyOrderTotalsByYear AS Cur
    LEFT OUTER JOIN @MyOrderTotalsByYear AS Prv
        ON Cur.orderyear = Prv.orderyear + 1;
```

Этот код вернет следующий результат.

orderyear	curyearqty	prvyearqty
2006	9581	NULL
2007	25489	9581
2008	16247	25489

Нужно отметить, что в SQL Server 2012 появился более эффективный способ решения этой задачи, который подразумевает использование функции LAG (см. ниже).

```
DECLARE @MyOrderTotalsByYear TABLE
(
    orderyear INT NOT NULL PRIMARY KEY,
    qty       INT NOT NULL
);

INSERT INTO @MyOrderTotalsByYear(orderyear, qty)
SELECT
    YEAR(O.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
        ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);

SELECT orderyear, qty AS curyearqty,
    LAG(qty) OVER(ORDER BY orderyear) AS prvyearqty
FROM @MyOrderTotalsByYear;
```

## Табличные типы

Табличные типы поддерживаются в SQL Server 2008 и SQL Server 2012. Чтобы создать табличный тип, в БД необходимо сохранить определение таблицы, на основе которого впоследствии будут объявляться табличные переменные. Позже его можно будет использовать во входящих параметрах для хранимых процедур и пользовательских функций.

К примеру, следующий код создает в текущей БД табличный тип под названием `dbo.OrderTotalsByYear`.

```
IF TYPE_ID('dbo.OrderTotalsByYear') IS NOT NULL
    DROP TYPE dbo.OrderTotalsByYear;

CREATE TYPE dbo.OrderTotalsByYear AS TABLE
(
    orderyear INT NOT NULL PRIMARY KEY,
    qty INT NOT NULL
);
```

Теперь, если вам нужно объявить табличную переменную, вы можете просто указать ее тип (в нашем случае `dbo.OrderTotalsByYear`), не записывая все определение целиком.

```
DECLARE @MyOrderTotalsByYear AS dbo.OrderTotalsByYear;
```

Давайте рассмотрим более сложный пример. Объявим переменную `@MyOrderTotalsByYear`, используя наш новый тип. Выполним запрос к таблицам `Orders` и `OrderDetails`, вычислим общее количество товара, заказанного в каждом году, сохраним результат в переменной и выведем ее содержимое.

```

DECLARE @MyOrderTotalsByYear AS dbo.OrderTotalsByYear;

INSERT INTO @MyOrderTotalsByYear(orderyear, qty)
SELECT
    YEAR(O.orderdate) AS orderyear,
    SUM(OD.qty) AS qty
FROM Sales.Orders AS O
    JOIN Sales.OrderDetails AS OD
        ON OD.orderid = O.orderid
GROUP BY YEAR(orderdate);

SELECT orderyear, qty FROM @MyOrderTotalsByYear;

```

Результирующий набор будет выглядеть так.

orderyear	qty
2006	9581
2007	25489
2008	16247

Табличные типы не просто делают код более компактным. Как я уже говорил, SQL Server позволяет указывать их в качестве типов входящих параметров для хранимых процедур и функций, что может оказаться чрезвычайно полезным.

## Динамические возможности языка SQL

SQL Server позволяет создавать пакеты в виде символьных строк, которые можно выполнять как обычный код. Для динамического выполнения кода существует два инструмента: команда `EXEC` (сокращенно от `EXECUTE`) и хранимая процедура `sp_executesql`. Разницу между ними, а также способы их применения рассмотрим ниже.

Динамические возможности языка SQL используются в разных целях.

- Автоматизация административных задач. Это может быть, например, извлечение метаданных или выполнение команды `BACKUP DATABASE` для каждой отдельной БД в рамках локального экземпляра SQL Server.
- Улучшение производительности некоторых участков кода. Например, создание произвольных параметризованных запросов, которые могут использовать предварительно заэкшированные планы выполнения (об этом чуть позже).
- Создание элементов кода с учетом данных, полученных в результате запроса. К примеру, оператор `PIVOT` можно сформировать динамически, если заранее неизвестно, какие элементы должны быть указаны в его инструкции `IN`.



### ПРИМЕЧАНИЕ

Будьте осторожны, формируя запросы на основе пользовательского ввода. Злоумышленники могут внедрить в ваше приложение вредоносный код. Лучше откажитесь от выполнения строковых данных, которые приходят извне (например, посредством параметров) или по крайней мере, выполняйте тщательную

проверку и старайтесь отслеживать попытки внедрения SQL-кода. В электронном справочнике есть хорошая статья на эту тему — «Атака SQL Injection».

## Команда EXEC

Первым инструментом для динамического выполнения кода, который появился в языке T-SQL, была команда EXEC. В качестве входящего аргумента она принимает символьную строку, которая потом выполняется в виде единого пакета. Поддерживаются форматы ASCII и Unicode.

Ниже представлен SQL-код с инструкцией PRINT, который хранится в строковой переменной @sql и выполняется с помощью команды EXEC.

```
DECLARE @sql AS VARCHAR(100);  
SET @sql = 'PRINT ''Это сообщение было напечатано динамическим  
SQL-пакетом.''';  
EXEC (@sql);
```

Обратите внимание, что одинарные кавычки внутри строки должны быть удвоены. Ниже показан результат.

Это сообщение было напечатано динамическим SQL-пакетом.

## Хранимая процедура sp\_executesql

Вслед за командой EXEC в SQL Server появилась поддержка хранимой процедуры sp\_executesql. Она более безопасная и гибкая, так как ее интерфейс заключается в поддержке входящих и исходящих параметров. Пакеты с кодом, которые передаются этой процедуре, должны иметь формат Unicode.

Предоставленная возможность использовать входящие и исходящие параметры при формировании динамического кода позволяет создавать куда более безопасные и эффективные запросы. Параметры, которые передаются извне, не могут быть интерпретированы как часть кода — их реально использовать только в качестве операндов в рамках выражения. Благодаря этому легко избежать атак типа SQL injection.

Хранимая процедура sp\_executesql может иметь более высокую производительность по сравнению с командой EXEC, поскольку поддержка параметризации помогает при многократном использовании закешированных планов выполнения. Такой план автоматически генерируется для каждого запроса; в нем описывается набор инструкций, которые должны применяться к объекту, порядок их обработки, список индексов и способ их использования, тип алгоритма соединения данных и т. д. Одно из условий повторного использования ранее закешированного плана заключается в том, что строка запроса должна совпадать с той, которая хранится в кэше. Хранимые процедуры с параметрами позволяют наиболее эффективно использовать имеющиеся планы выполнения, ведь даже если параметры изменятся, строка запроса останется прежней. Если вы по каким-то причинам решили не сохранять свой код в БД, у вас все равно остается возможность применять процедуру sp\_executesql, повышая тем самым вероятность многократного использования планов выполнения.



Процедура `sp_executesql` имеет два входящих значения (оба в формате Unicode) и блок, в котором они присваиваются. Первое значение, `@stmt` — это строка с кодом, который вы хотите выполнить. Во втором значении, `@params`, передаются входящие и исходящие параметры, которые разделяются запятыми и инициализируются в следующем блоке.

Ниже представлен пример динамического создания запроса, направленного к таблице **Sales.Orders**. Входящий параметр `@orderid` используется для фильтрации результатов.

```
DECLARE @sql AS NVARCHAR(100);

SET @sql = N'SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE orderid = @orderid;';

EXEC sp_executesql
    @stmt = @sql,
    @params = N'@orderid AS INT',
    @orderid = 10248;
```

Результирующий набор будет выглядеть так.

orderid	custid	empid	orderdate
10248	85	5	2006-07-04 00:00:00.000

Этот запрос присваивает входящему параметру число 10 248, но даже если вы передадите какое-то другое значение, строка с кодом останется прежней. Таким образом увеличивается вероятность повторного использования плана выполнения, который был закэширован ранее.

## Оператор PIVOT

Этот раздел не является обязательным к прочтению. Он ориентирован на читателей, которые уверенно владеют техникой разворачивания данных и хорошо усвоили основы динамического выполнения кода.

С оператором `PIVOT` вы познакомились в главе 7. Ранее я уже упоминал, что при написании статического запроса необходимо заранее знать, какие значения должны находиться в инструкции `IN`. Разворачивание данных в рамках статического кода продемонстрировано ниже.

```
SELECT *
FROM (SELECT shipperid, YEAR(orderdate) AS orderyear, freight
      FROM Sales.Orders) AS D
PIVOT(SUM(freight) FOR orderyear IN([2006],[2007],[2008])) AS P;
```

В данном примере мы обращаемся к таблице **Sales.Orders** и разворачиваем ее содержимое так, чтобы идентификаторы поставщиков были размещены по вертикали, годы следовали по горизонтали, а на пересечении находился общий годичный объем товара для каждого поставщика. Этот код возвращает следующий результат.

shipperid	2006	2007	2008
3	4233,78	11413,35	4865,38
1	2297,42	8681,38	5206,53
2	3748,67	12374,04	12122,14

При создании статического запроса нужно знать все значения, которые будут указаны в инструкции IN (в нашем случае это годы). Вам придется ежегодно изменять свой код. Но вы можете получить нужные значения и из самой таблицы, а затем передать их в пакет с динамическим SQL-кодом.

```

DECLARE
    @sql AS NVARCHAR(1000),
    @orderyear AS INT,
    @first AS INT;

DECLARE C CURSOR FAST_FORWARD FOR
    SELECT DISTINCT(YEAR(orderdate)) AS orderyear
    FROM Sales.Orders
    ORDER BY orderyear;

SET @first = 1;

SET @sql = N'SELECT *
FROM (SELECT shipperid, YEAR(orderdate) AS orderyear, freight
      FROM Sales.Orders) AS D
      PIVOT(SUM(freight) FOR orderyear IN(';

OPEN C;

FETCH NEXT FROM C INTO @orderyear;

WHILE @@fetch_status = 0
BEGIN
    IF @first = 0
        SET @sql = @sql + N',';
    ELSE
        SET @first = 0;

    SET @sql = @sql + QUOTENAME(@orderyear);

    FETCH NEXT FROM C INTO @orderyear;
END

CLOSE C;

DEALLOCATE C;

SET @sql = @sql + N')) AS P;';

EXEC sp_executesql @stmt = @sql;

```



### ПРИМЕЧАНИЕ

В более эффективных способах объединения строк используется курсор, например агрегатные функции среды CLR (Common Language Runtime) или параметр FOR XML PATH. Но это уже углубленный материал.

## Процедуры

**Процедуры** — это программируемые объекты, которые инкапсулируют код, предназначенный для вычисления результата или выполнения каких-то задач. SQL Server поддерживает три вида таких объектов: пользовательские функции, хранимые процедуры и триггеры.

Процедуры можно создавать как на языке T-SQL, так и посредством кода Microsoft .NET, используя интеграцию с CLR. Поскольку эта книга посвящена структурированному языку запросов, в своих примерах мы будем использовать первый вариант. К тому же T-SQL лучше подходит для решения задач, связанных с изменением данных. Платформа .NET была бы более уместной в случае применения итеративной логики, манипуляций со строками или интенсивных вычислительных операций.

### Пользовательские функции

**Пользовательские функции** инкапсулируют код, предназначенный для вычисления и возвращения определенного результата (с возможностью использования входящих параметров).

В SQL Server пользовательские функции могут возвращать как скалярные, так и табличные значения. Любую функцию реально разместить внутри запроса — либо вместо скалярного выражения, либо в рамках инструкции FROM (в случае с табличными значениями). В этом разделе мы рассмотрим пример со скалярной функцией.

У пользовательских функций не должно быть никаких побочных эффектов. Они не способны изменять структуру или содержимое БД. Это ограничение не столь очевидно, как может показаться. Например, вызов функций RAND (возвращает случайное значение) или NEWID (возвращает глобальный уникальный идентификатор GUID) тоже не проходит бесследно. Если функции RAND не передать начальное значение, SQL Server сгенерирует его автоматически, основываясь на предыдущем вызове; следовательно, промежуточные результаты должны храниться в БД. То же самое касается функции NEWID: система должна сохранять определенную информацию, которая будет использоваться при следующем вызове. Таким образом, вы не можете делать вызовы RAND и NEWID в своих пользовательских функциях.

Код, представленный ниже, создает пользовательскую функцию под названием dbo.GetAge, которая возвращает возраст человека на момент наступления определенной даты (аргумент @eventdate). День рождения тоже передается в виде входящего параметра (@eventdate).

```
IF OBJECT_ID('dbo.GetAge') IS NOT NULL DROP FUNCTION dbo.GetAge;
GO

CREATE FUNCTION dbo.GetAge
(
    @birthdate AS DATE,
    @eventdate AS DATE
)
```

```

RETURNS INT
AS
BEGIN
    RETURN
        DATEDIFF(year, @birthdate, @eventdate)
        - CASE WHEN 100 * MONTH(@eventdate) + DAY(@eventdate)
            < 100 * MONTH(@birthdate) + DAY(@birthdate)
            THEN 1 ELSE 0
        END;
END;
GO

```

Данная функция вычисляет возраст (измеряемый в годах) как разницу между днем рождения и заданной датой. Если переданные месяц и день меньше тех, что указаны в аргументе @birthdate, от результата отнимается один год. Выражение 100 \* month + day — это всего лишь прием, с помощью которого объединяются месяц и день. Например, для 12 февраля мы получим число 212.

Стоит отметить, что тело функции можно не ограничивать одной лишь командой RETURN. Оно может содержать элементы управления потоком выполнения, различные вычислительные операции и т. д. Наличие команды RETURN, которая возвращает итоговое значение, является обязательным.

Теперь рассмотрим применение пользовательских функций внутри запросов. Код, представленный ниже, обращается к таблице HR.Employees и вызывает на этапе SELECT функцию GetAge, которая вычисляет текущий возраст каждого сотрудника.

```

SELECT
    empid, firstname, lastname, birthdate,
    dbo.GetAge(birthdate, SYSDATETIME()) AS age
FROM HR.Employees;

```

Запустив этот код 24 марта 2014 г., вы бы получили следующий результат.

empid	firstname	lastname	birthdate	age
1	Сара	Дэвис	1958-12-08 00:00:00.000	55
2	Дон	Функ	1962-02-19 00:00:00.000	52
3	Джуди	Лью	1973-08-30 00:00:00.000	40
4	Иаиль	Пелед	1947-09-19 00:00:00.000	66
5	Свен	Бак	1965-03-04 00:00:00.000	49
6	Пол	Суурс	1973-07-02 00:00:00.000	40
7	Рассел	Кинг	1970-05-29 00:00:00.000	43
8	Мария	Камерон	1968-01-09 00:00:00.000	46
9	Зоя	Долгопятова	1976-01-27 00:00:00.000	38

(строк обработано: 9)

Очевидно, что значения, полученные для столбца age, будут зависеть от даты запуска этого запроса.

## Хранимые процедуры

**Хранимые процедуры** — это наборы кода на языке T-SQL, которые находятся на серверной стороне. Они поддерживают входящие и исходящие параметры, могут возвращать результирующие наборы и вызывать код, который имеет побочные эффекты. С их помощью легко изменять не только сами данные, но и их структуру.

Хранимые процедуры имеют множество преимуществ по сравнению с обычным кодом.

- Инкапсуляция логики. Изменения, которые вносятся в реализацию хранимой процедуры, производятся в одном месте, но доступны для всех ее пользователей.
- Улучшенное управление безопасностью. Вы можете выдавать полномочия на выполнение всей процедуры целиком, а не отдельных ее элементов. Представьте, что вам нужно разрешить пользователям удалять записи о клиентах, но при этом вы не хотите связывать эти полномочия непосредственно с таблицей **Customers**. Перед удалением клиенты должны проходить проверку на предмет открытых заказов или долгов. Все это вы можете делать внутри процедуры, выдавая полномочия на ее выполнение; при этом пользователь не будет иметь возможности удалять строки напрямую из таблицы. Кроме того, параметризованные хранимые процедуры позволяют предотвратить внедрение SQL-кода, особенно если использовать их вместо клиентских запросов.
- Возможность обработки ошибок и принятие необходимых мер внутри самой процедуры. Обработку ошибок мы обсудим позже в этой главе.
- Выигрыш в производительности. Выше я уже говорил о повторном использовании ранее заэкшированных планов выполнения. В хранимых процедурах в отличие от произвольного кода этот подход применяется по умолчанию. К тому же план процедуры не так быстро теряет свою актуальность. Еще одним преимуществом является уменьшение объема данных, передаваемых по сети. Клиентскому приложению нужно отправлять в SQL Server только имена хранимых процедур и их аргументы. Все промежуточные операции выполняются на сервере, а клиент уже получает готовый результат.

Возьмем в качестве простого примера следующий код, который создает хранимую процедуру под названием `Sales.GetCustomerOrders`; в качестве входящих аргументов она будет принимать идентификатор клиента (`@custid`) и диапазон дат (`@fromdate` и `@todate`). Процедура обращается к таблице `Sales.Orders` и возвращает заказы, размещенные указанным клиентом в определенном временном диапазоне; в результирующем наборе также содержится исходящий параметр `@numrows`, который представляет количество обработанных строк.

```
IF OBJECT_ID('Sales.GetCustomerOrders', 'P') IS NOT NULL
    DROP PROC Sales.GetCustomerOrders;
GO

CREATE PROC Sales.GetCustomerOrders
    @custid AS INT,
    @fromdate AS DATETIME = '19000101',
    @todate AS DATETIME = '99991231',
    @numrows AS INT OUTPUT
```

```
AS
SET NOCOUNT ON;

SELECT orderid, custid, empid, orderdate
FROM Sales.Orders
WHERE custid = @custid
      AND orderdate >= @fromdate
      AND orderdate < @todate;

SET @numrows = @@rowcount;
GO
```

Если при вызове процедуры опустить параметры @fromdate и @todate, SQL Server подставит значения по умолчанию (19000101 и 99991231). Чтобы сделать параметр @numrows исходящим, используется ключевое слово OUTPUT. С помощью команды SET NOCOUNT ON отключается сообщение о количестве обработанных строк (которое по умолчанию выводится элементами языка DML, в том числе SELECT).

Давайте вызовем эту хранимую процедуру, запросив информацию о заказах, которые клиент под номером 1 размещал в 2007 г. Значение параметра @numrows передается переменной @rc и возвращается в качестве результата, сигнализируя о том, сколько строк было обработано запросом.

```
DECLARE @rc AS INT;

EXEC Sales.GetCustomerOrders
    @custid = 1,
    @fromdate = '20070101',
    @todate = '20080101',
    @numrows = @rc OUTPUT;

SELECT @rc AS numrows;
```

Как видно по результату, нашим критериям соответствует три заказа.

orderid	custid	empid	orderdate
10643	1	6	2007-08-25 00:00:00.000
10692	1	4	2007-10-03 00:00:00.000
10702	1	4	2007-10-13 00:00:00.000

numrows
3

Запустите процедуру еще раз, указав идентификатор клиента, которого нет в таблице Orders (например, 100). Полученный вами результат будет свидетельствовать о том, что подходящих заказов нет.

orderid	custid	empid	orderdate

numrows
0

Конечно, это всего лишь простой пример. Хранимые процедуры позволяют делать намного больше.

## Триггеры

**Триггер** — это специальный вид хранимых процедур, который нельзя вызвать вручную. Он срабатывает и запускает свой код в момент, когда происходит определенное событие. SQL Server поддерживает два типа событий. Первый связан с операциями изменения данных (DML-триггеры), такими как INSERT, а второй относится к командам, которые определяют структуру БД (DDL-триггеры), например CREATE TABLE.

Триггеры используются для разных задач: это может быть проверка данных, обеспечение целостности, которое нельзя гарантировать путем ограничений, соблюдение различных правил и т. д.

Если к срабатыванию триггера привело событие, которое входит в транзакцию, то сам триггер тоже будет считаться частью этой транзакции. Если вызвать внутри него команду ROLLBACK TRAN, то все изменения, которые он успел выполнить, будут отменены.

В SQL Server срабатывание триггера связано с командами, а не с отдельными строками.

### DML-триггеры

DML-триггеры могут срабатывать либо после завершения соответствующего события, либо вместо него. В первом случае допускается использование только постоянных таблиц, а во втором можно дополнительно применить представления.

Таблицы, изменения в которых приводят к срабатыванию триггера, иногда помечаются ключевыми словами inserted или deleted. Первое позволяет получить новое содержимое строк, затронутых командами INSERT, MERGE или UPDATE; второе дает доступ к старым значениям, которые были актуальны перед выполнением команд DELETE, MERGE или UPDATE. Если триггер сработает вместо события, полученная вами таблица будет содержать строки, которые должны были быть затронуты в результате соответствующих операций.

Ниже показан пример простого триггера, который срабатывает после события и записывает сведения о добавленных строках. Запустите следующий код, чтобы создать в текущей БД две таблицы — dbo.T1 и dbo.T1\_Audit; первая будет проверяться, а вторая — содержать информацию о результатах проверки.

```
IF OBJECT_ID('dbo.T1_Audit', 'U') IS NOT NULL DROP TABLE dbo.T1_Audit;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;

CREATE TABLE dbo.T1
(
    keycol INT NOT NULL PRIMARY KEY,
    datacol VARCHAR(10) NOT NULL
);

CREATE TABLE dbo.T1_Audit
(
    audit_lsn INT NOT NULL IDENTITY PRIMARY KEY,
    dt DATETIME NOT NULL DEFAULT(SYSDATETIME()),
    login_name SYSNAME NOT NULL DEFAULT(ORIGINAL_LOGIN()),
```

```

    keycol      INT      NOT NULL,
    datacol     VARCHAR(10) NOT NULL
);

```

В таблице **T1\_Audit** есть столбец **audit\_lsn** со свойством **identity**, который хранит серийные номера. В столбец **dt** записываются дата и время добавления строк; для этого по умолчанию используется выражение **SYSDATETIME()**. Столбец **login\_name** представляет имена пользователей, которые добавляли строки (выражение по умолчанию **ORIGINAL\_LOGIN()**).

Теперь создадим для таблицы **T1** триггер **trg\_T1\_insert\_audit**, который будет отслеживать операции добавления.

```

CREATE TRIGGER trg_T1_insert_audit ON dbo.T1 AFTER INSERT
AS
SET NOCOUNT ON;

INSERT INTO dbo.T1_Audit(keycol, datacol)
SELECT keycol, datacol FROM inserted;
GO

```

Как видите, триггер просто добавляет в таблицу **T1\_Audit** результаты запроса, направленного к таблице **T1**. Если какой-то из столбцов не был заполнен вручную, в него записывается значение по умолчанию, сгенерированное вышеописанными выражениями. Чтобы увидеть наш триггер в действии, запустите следующий код.

```

INSERT INTO dbo.T1(keycol, datacol) VALUES(10, 'a');
INSERT INTO dbo.T1(keycol, datacol) VALUES(30, 'x');
INSERT INTO dbo.T1(keycol, datacol) VALUES(20, 'g');

```

Он будет срабатывать после каждой команды. Теперь выполните запрос к таблице **T1\_Audit**.

```

SELECT audit_lsn, dt, login_name, keycol, datacol
FROM dbo.T1_Audit;

```

Столбцы **dt** и **login\_name** содержат дату и время добавления строк; в столбце **login\_name** хранится имя пользователя, который подключался к SQL Server и выполнял эти операции.

audit_lsn	dt	login_name	keycol	datacol
1	2014-03-25 10:09:12.227	WIN81\troorl	10	a
2	2014-03-25 10:09:12.229	WIN81\troorl	30	x
3	2014-03-25 10:09:12.248	WIN81\troorl	20	g

Закончив с примерами, запустите следующий код, чтобы очистить БД.

```

IF OBJECT_ID('dbo.T1_Audit', 'U') IS NOT NULL DROP TABLE dbo.T1_Audit;
IF OBJECT_ID('dbo.T1', 'U') IS NOT NULL DROP TABLE dbo.T1;

```

### DDL-триггеры

SQL Server поддерживает DDL-триггеры, которые могут использоваться для таких задач, как проверка и управление процессом изменения данных, соблюдение



различных правил и т. д. Локальная версия SQL Server позволяет создавать DDL-триггеры на уровне БД или сервера в зависимости от того, в каком контексте произошло событие. SQL Database в настоящий момент поддерживает только уровень БД.

Примером события, которое находится в области видимости БД, может служить команда `CREATE TABLE`. Команда `CREATE DATABASE`, напротив, выполняется в контексте всего сервера. В SQL Server DDL-триггеры могут срабатывать только после определенного события, но не вместо.

Внутри триггера легко получить информацию о том, как он был вызван; для этого используется функция `EVENTDATA`, которая возвращает данные в формате XML. С помощью выражений XQuery извлекаются различные атрибуты, касающиеся события, такие как время возникновения, тип и имя пользователя.

Следующий код создает таблицу `dbo.AuditDDLEvents`, которая будет хранить сведения о событиях.

```
IF OBJECT_ID('dbo.AuditDDLEvents', 'U') IS NOT NULL
    DROP TABLE dbo.AuditDDLEvents;

CREATE TABLE dbo.AuditDDLEvents
(
    audit_lsn          INT          NOT NULL IDENTITY,
    posttime          DATETIME NOT NULL,
    eventtype         SYSNAME  NOT NULL,
    loginname         SYSNAME  NOT NULL,
    schemaname        SYSNAME  NOT NULL,
    objectname        SYSNAME  NOT NULL,
    targetobjectname  SYSNAME  NULL,
    eventdata         XML       NOT NULL,
    CONSTRAINT PK_AuditDDLEvents PRIMARY KEY(audit_lsn)
);
```

Обратите внимание на столбец `eventdata` типа XML. В него триггер сохраняет всю информацию о причине своего срабатывания.

Запустите следующий код, чтобы создать триггер под названием `trg_audit_ddl_events`. Он будет основан на группе событий `DDL_DATABASE_LEVEL_EVENTS`, которая представляет все события языка DDL на уровне БД.

```
CREATE TRIGGER trg_audit_ddl_events
    ON DATABASE FOR DDL_DATABASE_LEVEL_EVENTS
AS
SET NOCOUNT ON;

DECLARE @eventdata AS XML = eventdata();

INSERT INTO dbo.AuditDDLEvents(
    posttime, eventtype, loginname, schemaname,
    objectname, targetobjectname, eventdata)
VALUES(
    @eventdata.value('( /EVENT_INSTANCE/PostTime) [1] ',
    'VARCHAR(23) '),
    @eventdata.value('( /EVENT_INSTANCE/EventType) [1] ',      'SYSNAME'),
    @eventdata.value('( /EVENT_INSTANCE/LoginName) [1] ',      'SYSNAME'),
```

```
@eventdata.value('(/EVENT_INSTANCE/SchemaName)[1]','SYSNAME'),
@eventdata.value('(/EVENT_INSTANCE/ObjectName)[1]','SYSNAME'),
@eventdata.value('(/EVENT_INSTANCE/TargetObjectName)[1]','SYSNAME'),
@eventdata);

GO
```

Сначала вся информация о событии, полученная из функции EVENTDATA, сохраняется в переменной @eventdata. Затем в таблицу AuditDDLEvents добавляется новая строка, атрибуты которой заполняются с помощью выражений XQuery (а именно с использованием метода .value, вызываемого из события); последнему атрибуту присваиваются исходные данные в формате XML.

Код, представленный ниже, содержит несколько команд языка DDL. Запустите его, чтобы проверить наш триггер в работе.

```
CREATE TABLE dbo.T1(col1 INT NOT NULL PRIMARY KEY);
ALTER TABLE dbo.T1 ADD col2 INT NULL;
ALTER TABLE dbo.T1 ALTER COLUMN col2 INT NOT NULL;
CREATE NONCLUSTERED INDEX idx1 ON dbo.T1(col2);
```

Теперь извлеките содержимое таблицы AuditDDLEvents.

```
SELECT * FROM dbo.AuditDDLEvents;
```

Результат показан ниже (для удобства он разбит на две части). Нужно отметить, что атрибуты posttime и loginname должны отражать время и имя пользователя, актуальные для вашей системы.

audit_lsn	posttime	eventtype	loginname
1	2014-03-25 10:52:12.213	CREATE_TABLE	WIN81\troorl
2	2014-03-25 10:52:12.506	ALTER_TABLE	WIN81\troorl
3	2014-03-25 10:52:12.509	ALTER_TABLE	WIN81\troorl
4	2014-03-25 10:52:12.509	CREATE_INDEX	WIN81\troorl

audit_lsn	schemaname	objectname	targetobjectname	eventdata
1	dbo	T1	NULL	<EVENT_INSTANCE>...
2	dbo	T1	NULL	<EVENT_INSTANCE>...
3	dbo	T1	NULL	<EVENT_INSTANCE>...
4	dbo	idx1	T1	<EVENT_INSTANCE>...

Закончив с примерами, запустите следующий код, чтобы очистить БД.

```
DROP TRIGGER trg_audit_ddl_events ON DATABASE;
DROP TABLE dbo.AuditDDLEvents;
```

## Обработка ошибок

SQL Server предоставляет средства для обработки ошибок, например конструкцию языка T-SQL под названием TRY . . . CATCH, ее мы рассмотрим в первую очередь. Чуть позже речь пойдет о целом ряде функций, которые позволяют получить различную информацию о произошедшей ошибке.

В блоке TRY (точнее, между ключевыми словами BEGIN TRY и END TRY) размещается обычный код, который может привести к ошибкам. Смежный блок CATCH (ключевые слова BEGIN CATCH и END CATCH) предназначен для обработки нештатных ситуаций. При отсутствии ошибок блок CATCH просто пропускается. Он выполняется только в том случае, если в блоке TRY случается непредвиденная ситуация. Стоит отметить, что ошибка, которая была перехвачена и обработана конструкцией TRY . . . CATCH, не доходит до внешнего кода.

Ниже проиллюстрирована ситуация, когда блок TRY работает в штатном режиме.

```
BEGIN TRY
    PRINT 10/2;
    PRINT 'Нет ошибок';
END TRY
BEGIN CATCH
    PRINT 'Ошибка';
END CATCH;
```

Весь код в блоке TRY завершился успешно; следовательно, блок CATCH был пропущен. Результат показан ниже.

```
5
Нет ошибок
```

Теперь выполним похожий код, но с делением на ноль, чтобы вызвать ошибку.

```
BEGIN TRY
    PRINT 10/0;
    PRINT 'Нет ошибок';
END TRY
BEGIN CATCH
    PRINT 'Ошибка';
END CATCH;
```

Когда в первой команде PRINT произойдет ошибка — деление на ноль, поток выполнения перейдет к соответствующему блоку CATCH. При этом вторая команда PRINT не выполняется. В итоге будет сгенерирован следующий результат.

```
Ошибка
```

Обычно блок CATCH используется, чтобы выяснить причину ошибки и принять необходимые меры. В связи с этим в SQL Server предусмотрено несколько функций. Пожалуй, наиболее важной из них является функция ERROR\_NUMBER, которая возвращает целочисленный код ошибки. Как правило, именно на основе ее значения предпринимаются те или иные меры. Функция ERROR\_MESSAGE возвращает текст сообщения об ошибке. Список всех кодов и сообщений можно найти в каталоге представления sys.messages. Функции ERROR\_SEVERITY и ERROR\_STATE возвращают степень важности ошибки и ее состояние. Функция ERROR\_LINE возвращает номер строки, ставшей источником проблемы. Наконец функция ERROR\_PROCEDURE позволяет узнать имя процедуры, в которой возникла ошибка; если же сбой случился за пределами процедуры, вы получите NULL.

Чтобы подробнее рассмотреть процесс обработки ошибок, в том числе с использованием вышеприведенных функций, создадим в текущей БД таблицу под названием `dbo.Employees`.

```
IF OBJECT_ID('dbo.Employees') IS NOT NULL DROP TABLE dbo.Employees;
CREATE TABLE dbo.Employees
(
    empid    INT            NOT NULL,
    empname  VARCHAR(25) NOT NULL,
    mgrid    INT            NULL,
    CONSTRAINT PK_Employees PRIMARY KEY(empid),
    CONSTRAINT CHK_Employees_empid CHECK(empid > 0),
    CONSTRAINT FK_Employees_Employees
        FOREIGN KEY(mgrid) REFERENCES dbo.Employees(empid)
);
```

Следующий код содержит блок `TRY`, внутри которого происходит добавление новой строки в таблицу `Employees`. Для определения типа ошибки в блоке `CATCH` используется сочетание функции `ERROR_NUMBER` и элементов управления потоком выполнения. Все ошибки, которые не были идентифицированы, генерируются заново.



#### ПРИМЕЧАНИЕ

Возможность повторного генерирования ошибки с помощью команды `THROW` была добавлена в SQL Server 2012.

Здесь также выводятся значения остальных функций типа `ERROR_*`; так вы сами сможете увидеть, какая информация об ошибках вам доступна.

```
BEGIN TRY

    INSERT INTO dbo.Employees(empid, empname, mgrid)
        VALUES(1, 'Emp1', NULL);
    -- Попробуйте присвоить empid значения 0, 'A', NULL

END TRY
BEGIN CATCH

    IF ERROR_NUMBER() = 2627
    BEGIN
        PRINT 'Обрабатываем нарушения первичного ключа...';
    END
    ELSE IF ERROR_NUMBER() = 547
    BEGIN
        PRINT 'Обрабатываем нарушения ограничений CHECK/FK...';
    END
    ELSE IF ERROR_NUMBER() = 515
    BEGIN
        PRINT 'Обрабатываем нарушение ограничения NOT NULL...';
    END
    ELSE IF ERROR_NUMBER() = 245
    BEGIN
        PRINT 'Обрабатываем ошибку приведения типов...';
    END
    ELSE
```

```
BEGIN
    PRINT ' Генерируем ошибку еще раз...';
    THROW; -- SQL Server 2012 only
END

PRINT ' Номер : '      + CAST(ERROR_NUMBER() AS VARCHAR(10));
PRINT ' Сообщение : ' + ERROR_MESSAGE();
PRINT ' Степень важности: '
      + CAST(ERROR_SEVERITY() AS VARCHAR(10));
PRINT ' Состояние : ' + CAST(ERROR_STATE() AS VARCHAR(10));
PRINT ' Строка : '    + CAST(ERROR_LINE() AS VARCHAR(10));
PRINT ' Процедура : ' + COALESCE(ERROR_PROCEDURE(), 'За пределами
процедуры');

END CATCH;
```

При первом запуске строка успешно добавляется в таблицу **Employees**, поэтому блок **CATCH** пропускается. Вы получите следующий результат.

```
(строк обработано: 1)
```

При втором запуске в команде **INSERT** происходит сбой, поток выполнения переходит к блоку **CATCH**, где обнаруживается нарушение ограничений первичного ключа. Результат будет выглядеть так.

```
Обрабатываем нарушения первичного ключа...
Номер : 2627
Сообщение : Нарушено "PK_Employees" ограничения PRIMARY KEY. Не удается
вставить повторяющийся ключ в объект "dbo.Employees". Повторяющееся
значение ключа: (1).
Степень важности: 14
Состояние : 1
Строка : 3
Процедура : За пределами процедуры
```

Если вы хотите увидеть другие ошибки, попробуйте передать в качестве идентификатора сотрудника значения 0, 'A', NULL.

Чтобы продемонстрировать реакцию на возникающие ошибки, я использовал команду **PRINT**. Но обычно под обработкой ошибок понимают не просто вывод информации о них.

Вы можете создать хранимую процедуру, которая будет инкапсулировать код из предыдущего примера.

```
IF OBJECT_ID('dbo.ErrInsertHandler', 'P') IS NOT NULL
    DROP PROC dbo.ErrInsertHandler;
GO

CREATE PROC dbo.ErrInsertHandler
AS
SET NOCOUNT ON;

IF ERROR_NUMBER() = 2627
BEGIN
    PRINT ' Обрабатываем нарушения первичного ключа...';
END
```

```

ELSE IF ERROR_NUMBER() = 547
BEGIN
    PRINT ' Обрабатываем нарушения ограничений CHECK/FK...';
END
ELSE IF ERROR_NUMBER() = 515
BEGIN
    PRINT ' Обрабатываем нарушение ограничения NOT NULL...';
END
ELSE IF ERROR_NUMBER() = 245
BEGIN
    PRINT ' Обрабатываем ошибку приведения типов...';
END

PRINT ' Номер : '      + CAST(ERROR_NUMBER() AS VARCHAR(10));
PRINT ' Сообщение : ' + ERROR_MESSAGE();
PRINT ' Степень важности: '
      + CAST(ERROR_SEVERITY() AS VARCHAR(10));
PRINT ' Состояние : ' + CAST(ERROR_STATE() AS VARCHAR(10));
PRINT ' Строка : '    + CAST(ERROR_LINE() AS VARCHAR(10));
PRINT ' Процедура : ' + COALESCE(ERROR_PROCEDURE(), 'За пределами
процедуры');
GO

```

Теперь при получении одной из тех ошибок, которые вы хотите обрабатывать локально, можно будет просто запустить в блоке `CATCH` хранимую процедуру; в противном случае ошибка может быть сгенерирована повторно.

```

BEGIN TRY

    INSERT INTO dbo.Employees(empid, empname, mgrid)
        VALUES(1, 'Emp1', NULL);

END TRY
BEGIN CATCH

    IF ERROR_NUMBER() IN (2627, 547, 515, 245)
        EXEC dbo.ErrInsertHandler;
    ELSE
        THROW;

END CATCH;

```

Так вы получите код, пригодный для многократного использования; к тому же вся обработка ошибок будет находиться в одном месте.

## В заключение

В данной главе вы познакомились с основными аспектами программируемых объектов, узнали, какими возможностями обладает SQL Server в этой области, а также создали несколько собственных программных модулей. Мы рассмотрели довольно обширный список тем: переменные, пакеты, элементы управления потоком выполнения, курсоры, временные таблицы, динамические средства языка SQL, пользовательские функции, хранимые процедуры, триггеры и обработку ошибок. Надеюсь, главным приоритетом для вас были общие принципы и возможности, а не мелкие детали в примерах кода.

# Приложение

## ПРИСТУПАЕМ К РАБОТЕ

Данное дополнение должно помочь подготовиться к работе с этой книгой, чтобы вы могли извлечь из нее максимум пользы.

Все примеры кода, которые здесь приводятся, можно запускать на локальном (коробочном) экземпляре Microsoft SQL Server; впрочем, большинство из них будут работать и в облачном сервисе (Windows Azure SQL Database — бывший SQL Azure). Различия между этими версиями подробно описываются в разделе «Разновидности SQL Server» в главе 1.

В первом разделе приводятся адреса сайтов, на которых вы сможете найти информацию о том, с чего следует начинать работу с SQL Database.

Во втором разделе мы будем исходить из того, что вы хотите запускать примеры кода на локальном экземпляре SQL Server. Пошагово рассмотрим процесс установки SQL Server 2012. Если у вас уже имеется установленная копия, можете смело перелистывать дальше.

В третьем разделе указаны ссылки на исходный код, который прилагается к этой книге. Там же содержатся инструкции относительно создания демонстрационной БД на примере SQL Server и SQL Database.

В четвертом разделе рассказывается о том, как писать и запускать код на языке T-SQL, используя SQL Server и SQL Server Management Studio (SSMS).

В самом конце мы поговорим об электронном справочнике и его роли в изучении языка T-SQL.

## Начинаем работу с SQL Database

Если вы хотите запускать приводимые здесь примеры в SQL Database, вам понадобятся соответствующие сервер и учетная запись, которая позволит создавать БД (также можно попросить администратора создать для вас демонстрационную БД). Если у вас все еще нет доступа к SQL Database, вам пригодится информация, размещенная на главной странице сервиса Windows Azure ([windowsazure.com](http://windowsazure.com)).

Прежде всего, вам нужна учетная запись Windows Live ID, которую легко создать по адресу [signup.live.com](http://signup.live.com). Подписавшись на сервис Windows Azure, вы сможете подключиться к панели администрирования Windows Azure Platform Management Portal, которая позволяет управлять серверами и БД.

На главной странице сервиса Windows Azure предлагается несколько вариантов подписки — платная и пробная (временно бесплатная). Там вы сможете получить доступ к целому ряду разделов, в частности к панели управления, ресурсам сообщества и технической поддержки.

Подключившись к SQL Database, перейдите к инструкциям по загрузке исходного кода и созданию демонстрационной БД, которые приводятся в третьем разделе.

## Установка локального экземпляра SQL Server

Этот раздел предназначен для тех, кто хочет запускать приводимые здесь примеры и упражнения в рамках локального экземпляра SQL Server (доступа к которому, как мы предполагаем, у вас все еще нет). Вы можете использовать любой вариант SQL Server 2012; исключение составляет версия SQL Server Compact, которая, в отличие от остальных, не имеет полноценной поддержки языка T-SQL. Ниже представлены инструкции, которые помогут вам загрузить и установить свою копию SQL Server.

### 1. Загрузка SQL Server

Как я уже упоминал, для работы с материалом, который содержится в этой книге, подойдет любой вариант SQL Server 2012, кроме SQL Server Compact. Подписка на сервисы MSDN (Microsoft Developer Network) дает бесплатный доступ к версии SQL Server 2012 Developer (только в целях обучения); загрузить ее можно на странице [msdn.microsoft.com/ru-ru/sqlserver/default.aspx](http://msdn.microsoft.com/ru-ru/sqlserver/default.aspx).

Пробная версия SQL Server 2012 доступна по адресу [www.microsoft.com/sqlserver/ru/ru/get-sql-server/try-it.aspx](http://www.microsoft.com/sqlserver/ru/ru/get-sql-server/try-it.aspx). В этом разделе мы будем использовать второй вариант.

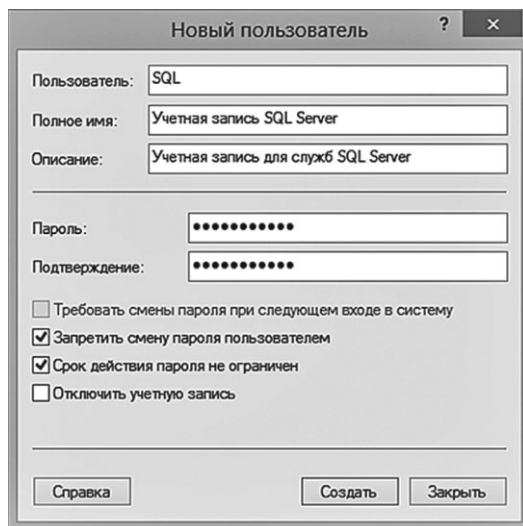
### 2. Создание учетной записи

Для начала вам необходимо создать учетную запись, которая в дальнейшем будет использоваться для работы со службами SQL Server.

Чтобы создать учетную запись, выполните следующее.

- 1) Щелкните правой кнопкой мыши на значке **Мой компьютер** и выберите пункт меню **Управление**, чтобы открыть утилиту для управления компьютером.
- 2) Перейдите к разделу **Управление компьютером (локальным) ► Служебные программы ► Локальные пользователи и группы ► Пользователи**.
- 3) Щелкните правой кнопкой мыши на папке **Пользователи** и выберите пункт меню **Новый пользователь**.
- 4) Откроется окно диалога **Новый пользователь**, где вы сможете указать информацию о новой учетной записи (рис. П.1).





**Рис. П.1.** Окно диалога для добавления нового пользователя

- 4.1. Введите имя пользователя (например, SQL) и безопасный пароль (который нужно подтвердить). При желании также можно ввести полное имя пользователя (например, Учетная запись SQL Server) и описание (например, Учетная запись для служб SQL Server).
- 4.2. Снимите флажок **Требовать смены пароля при следующем входе в систему**.
- 4.3. Установите флажки **Запретить смену пароля пользователем** и **Срок действия пароля не ограничен**.
- 4.4. Нажмите кнопку **Создать**, чтобы создать новую учетную запись.

### 3. Установка необходимых инструментов

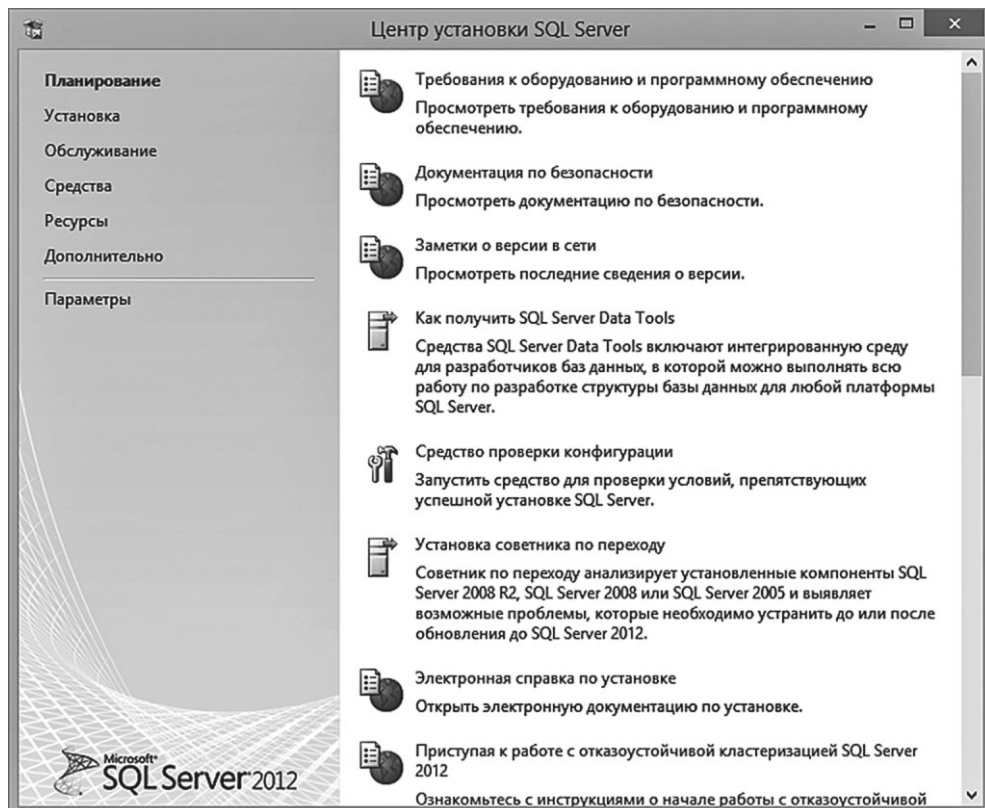
Теперь вы можете перейти в установочный каталог SQL Server и запустить программу `setup.exe`. Но прежде проверьте, установлены ли на вашем компьютере все необходимые инструменты: пакеты Microsoft .NET Framework 3.5 SP1 и the .NET Framework 4, а также обновления для Windows Installer. Если в вашей системе нет платформы .NET 3.5, программа установки сгенерирует ошибку и ссылку на страницу загрузки. Остальные инструменты будут установлены автоматически. Вероятно, вам придется перезагрузить компьютер и заново запустить процесс установки.

### 4. Установка ядра базы данных, документации и утилит

Установив весь необходимый инструментарий, вы можете приступить к установке SQL Server.

Выполните следующие действия, чтобы установить ядро БД, документацию и утилиты.

1. Теперь, когда ваша система готова, можете запустить программу `setup.exe`. Перед вами должно открыться диалоговое окно **Центр установки SQL Server** (рис. П.2).



**Рис. П.2.** Центр установки SQL Server

2. Выберите на левой панели пункт **Установка**. Содержимое окна изменится.
3. Выберите на правой панели пункт **Новая установка изолированного экземпляра SQL Server** или **добавление компонентов к существующей установке**. На экране появится диалоговое окно **Правила поддержки установки**.
4. Нажмите кнопку **Показать подробности**, чтобы увидеть состояние правил поддержки (рис. П.3). Убедитесь, что в ходе проверки не было обнаружено никаких проблем.
5. Закончив с проверкой, нажмите кнопку **ОК**. На экране появится диалоговое окно **Ключ продукта** (рис. П.4).

Стоит отметить, что в некоторых ситуациях диалоговые окна **Установка файлов установки** и **Правила поддержки установки** могут появиться перед вводом ключа. В этом случае вам нужно просто следовать инструкциям в пунктах 7–9.

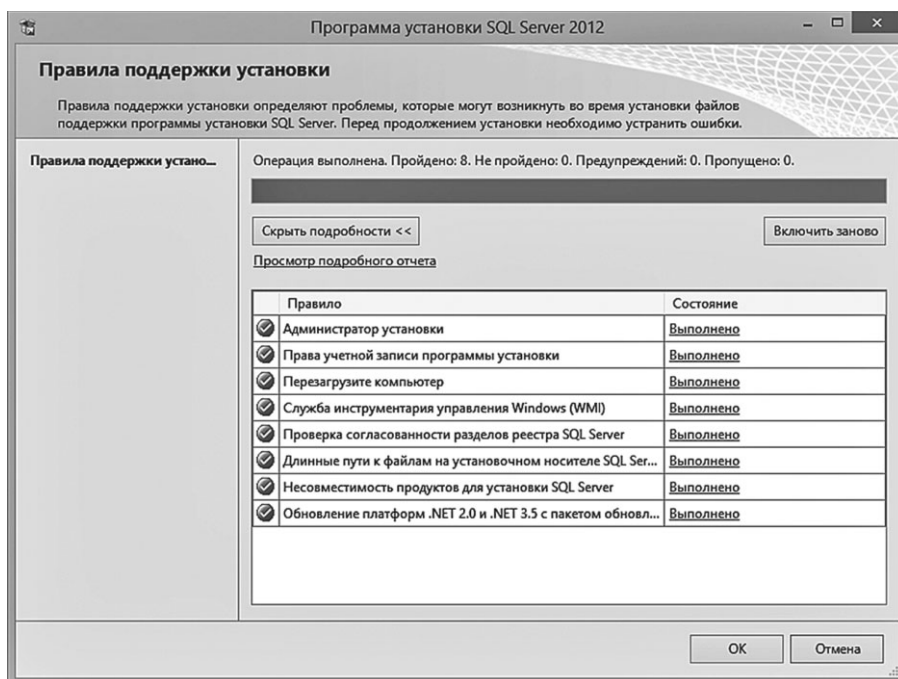


Рис. П.3. Диалоговое окно Правила поддержки установки

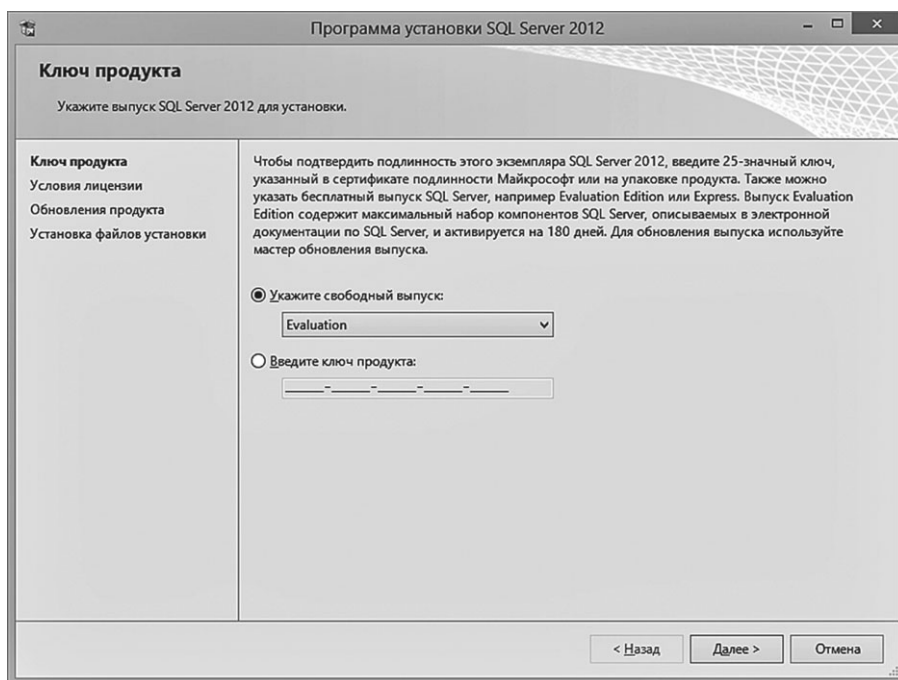


Рис. П.4. Диалоговое окно Ключ продукта

- 6) Выберите в списке **Укажите свободный выпуск** пункт **Evaluation** и щелкните на кнопке **Далее**, чтобы продолжить. На экране появится диалоговое окно **Условия лицензионного соглашения**.
- 7) Подтвердите, что вы соглашаетесь с условиями, и нажмите кнопку **Далее**. Перед вами появится диалоговое окно **Установка файлов установки**.
- 8) Щелкните на кнопке **Установить**. На экране снова появится диалоговое окно **Правила поддержки установки**.
- 9) Нажмите кнопку **Показать подробности**, чтобы увидеть состояние правил поддержки и убедиться в отсутствии каких-либо проблем. Нажмите кнопку **Далее**, чтобы продолжить.

Перед вами появится диалоговое окно **Роль установки**. Оставьте выбранным пункт **Установка компонентов SQL Server** и щелкните на кнопке **Далее**. Вы увидите диалоговое окно **Выбор компонентов**.

Установите флажки так, как показано на рисунке П.5.

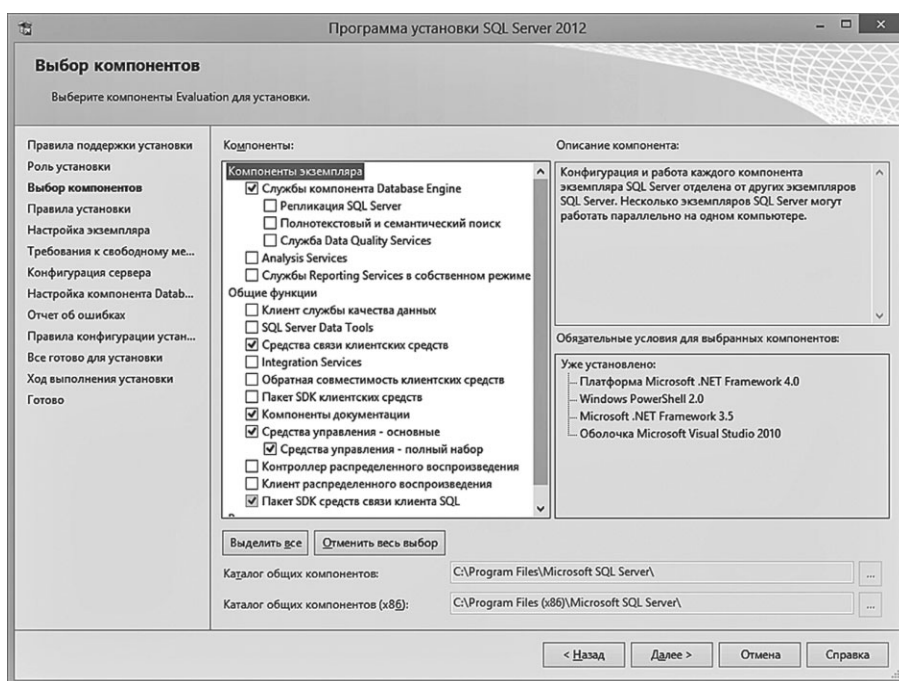


Рис. П.5. Диалоговое окно **Выбор компонентов**

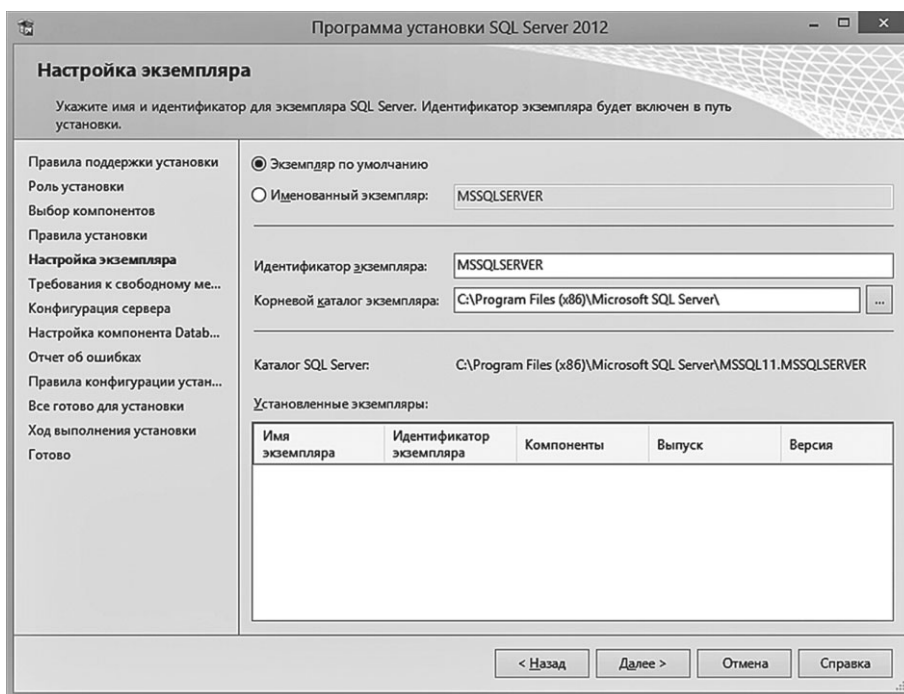
Выберите следующие пункты:

- Службы компонента Database Engine;
- Средства связи клиентских средств;

- Компоненты документации;
- Средства управления — основные.

Это все компоненты, которые необходимы для работы с данной книгой.

Закончив с этим, нажмите кнопку **Далее**. Вам нужно будет нажать ее еще раз, если на экране снова появится окно **Правила установки**. Вы перейдете к окну **Настройка экземпляра** (рис. П.6).



**Рис. П.6.** Диалоговое окно **Настройка экземпляра**

- Если на вашем компьютере еще нет установленной копии SQL Server, будет логично выбрать по умолчанию новый экземпляр; для этого просто оставьте выделенным вариант **Экземпляр по умолчанию**. Если вы хотите настроить именованный экземпляр, выберите соответствующий пункт и введите подходящее имя (например, SQL2012).

В первом случае для подключения к SQL Server достаточно ввести имя компьютера (например, WIN81), а во втором нужно дополнительно указать имя экземпляра (например, WIN81\SQL2012).

- Когда закончите, щелкните на кнопке **Далее**. Вы увидите диалоговое окно **Требования к свободному месту на диске**. Убедитесь в том, что у вас есть достаточно свободного места для установки.
- Нажмите кнопку **Далее**. На экране появится диалоговое окно **Конфигурация сервера**.

- 13) Введите имя и пароль созданной вами учетной записи для служб SQL Server Agent и SQL Server Database Engine, как это показано на рисунке П.7.

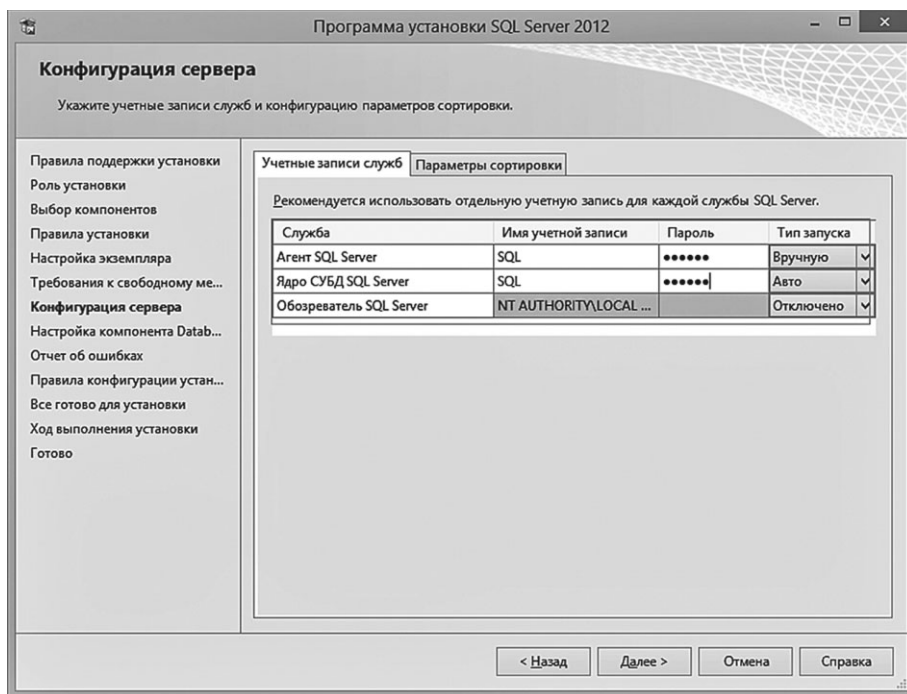


Рис. П.7. Диалоговое окно Конфигурация сервера

Конечно, ваша учетная запись не обязательно должна называться так, как в примере (SQL).

Для работы с примерами, которые приводятся в этой книге, вам не нужно изменять настройки во вкладке **Параметры сортировки**, но если вы хотите узнать больше о том, как выполняются операции сравнения, можете ознакомиться с разделом «Работа с символьными данными» в главе 2.

- 14) Щелкните на кнопке **Далее**, чтобы перейти к диалоговому окну **Настройка компонента Database Engine**.
- 15) Убедитесь, что в качестве режима проверки подлинности во вкладке **Конфигурация сервера** выбран вариант **Режим проверки подлинности Windows**. В разделе **Назначьте администраторов SQL Server** щелкните на кнопке **Добавить текущего пользователя**, чтобы указать текущую учетную запись с правами администратора (рис. П.8). Администраторы имеют неограниченный доступ к ядру базы данных SQL Server.

Конечно, вместо «WIN81\troorl» будет добавлено имя текущего пользователя.

Если вы хотите изменить директории для хранения данных, можете сделать это во вкладке **Каталоги данных**.

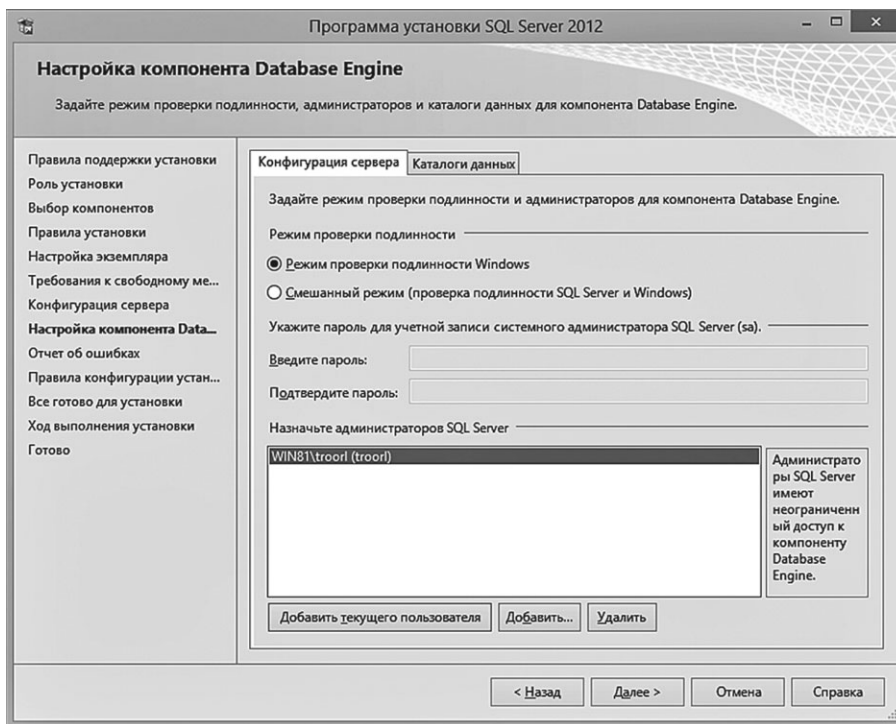


Рис. П.8. Диалоговое окно Настройка компонента Database Engine

- 16) Нажмите кнопку **Далее**. На экране появится диалоговое окно **Отчет об ошибках**. Выберите те настройки, которые вам больше подходят, и щелкните на кнопке **Далее**, чтобы перейти к диалоговому окну **Правила конфигурации установки**.
- 17) Нажмите кнопку **Показать подробности**, чтобы узнать состояние правил установки и убедиться в отсутствии каких-либо проблем. Нажав кнопку **Далее**, вы перейдете к окну **Все готово для установки**, где приводится список выбранных вами настроек.
- 18) Убедитесь в том, что список корректно отражает сделанный выбор, и нажмите кнопку **Установить**.

Начнется процесс установки, ход которого будет отображаться в соответствующем окне. Там помимо общего индикатора выполнения будет выводиться информация о каждом отдельном компоненте (рис. П.9). В конце будет выведено сообщение об успешном завершении процесса установки.

- 19) Нажмите кнопку **Далее**, чтобы перейти к диалоговому окну **Завершено** (рис. П.10). Это окно должно сообщить об успешном завершении установки.

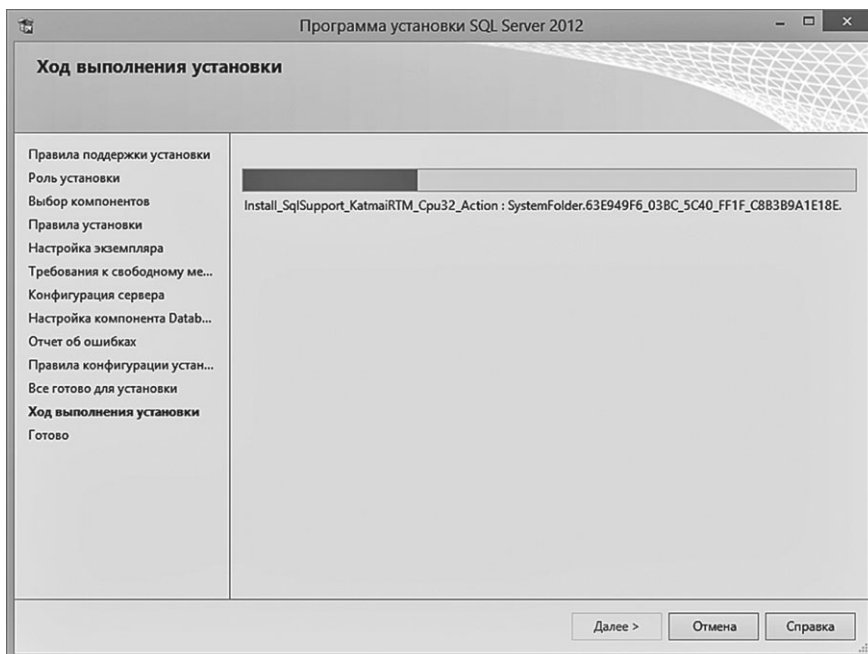


Рис. П.9. Диалоговое окно Ход выполнения установки

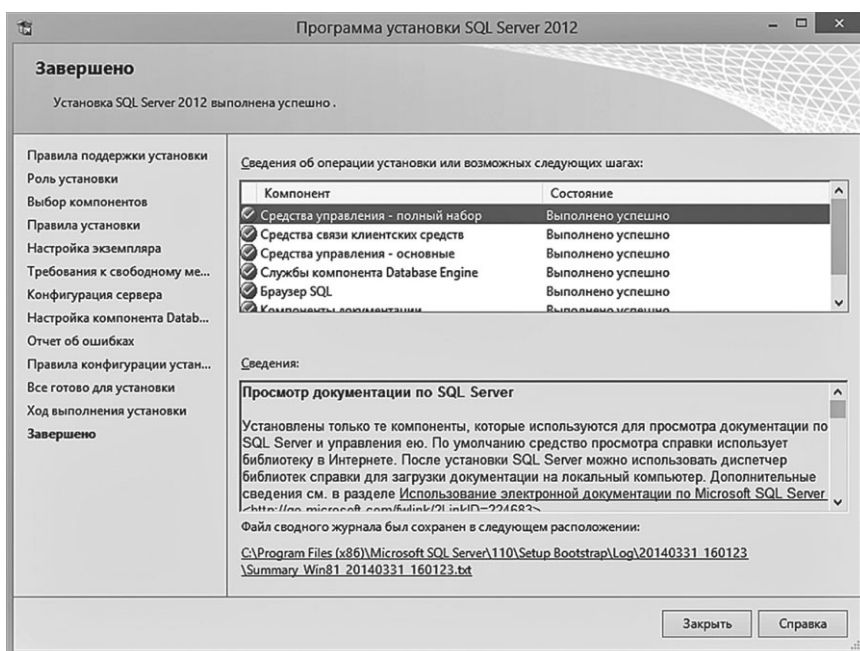


Рис. П.10. Диалоговое окно Завершено

20) Нажмите кнопку **Закрыть**, чтобы выйти из установочной программы.



## Загрузка исходного кода и подготовка демонстрационной базы данных

Инструкции по загрузке исходного кода можно найти здесь: [eksmo.ru/smv/SQLFundamentals.rar](http://eksmo.ru/smv/SQLFundamentals.rar). Там вы найдете ссылку на единый архив с примерами для этой книги, а также скрипт под названием **TSQL2012.sql**, с помощью которого можно создать демонстрационную БД. Распакуйте файлы в локальный каталог (например, C:\TSQlFundamentals).

С каждой главой связано не более трех файлов с расширением \*.sql. В первом (который называется так же, как глава) содержится исходный код примеров. Это сделано для удобства, чтобы вам не нужно было набирать все вручную. В остальных двух находятся упражнения и решения к ним; их можно легко определить по соответствующим названиям. Для открытия и запуска этих скриптов вам понадобится программа SQL Server Management Studio (SSMS). О том, как работать с SSMS, будет рассказано в следующем разделе.

В архиве также содержится текстовый файл под названием **orders.txt**, который пригодится при выполнении примеров в разделе «Изменение данных» в главе 8. Для создания демонстрационной БД **TSQL2012** (рис. П.11) предусмотрен скрипт **TSQL2012.sql**; запустите его, предварительно подключившись к SQL Server. Если у вас нет опыта запуска SQL-скриптов, можете воспользоваться инструкциями, которые приведены ниже.

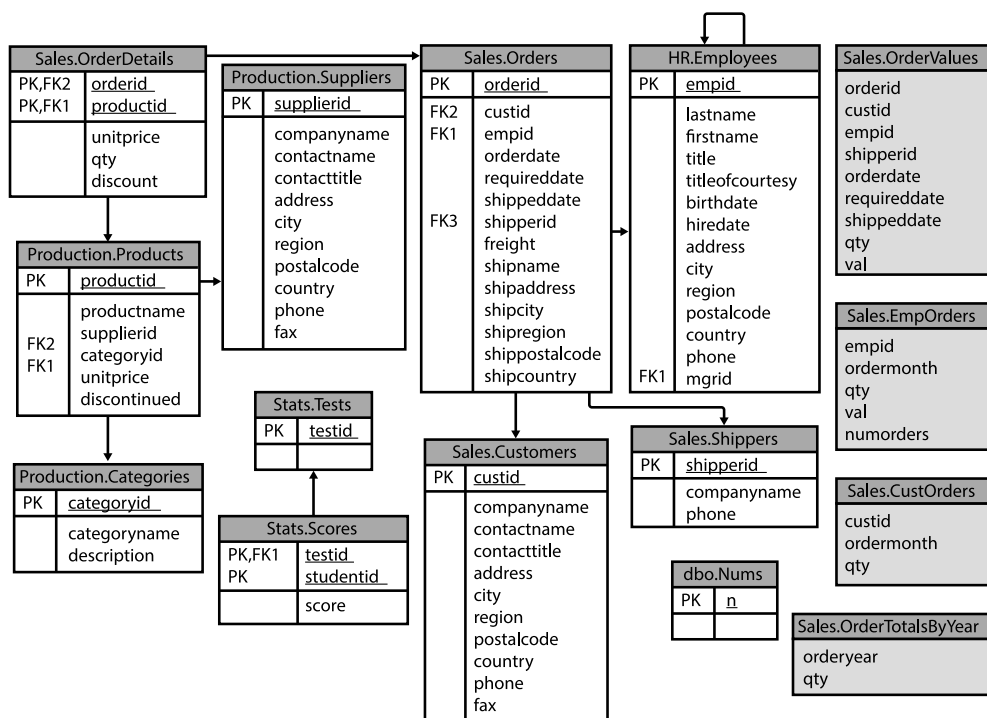


Рис. П.11. Структура базы данных TSQL2012

Чтобы создать и заполнить демонстрационную БД в рамках локального экземпляра SQL Server, выполните следующие действия.

1. Откройте приложение SSMS, дважды щелкнув на файле **TSQL2012.sql** в программе «Проводник». Перед вами появится окно **Подключиться к компоненту Database Engine**.
2. Убедитесь, что в поле ввода **Имя сервера** указано название того экземпляра, к которому вы хотите подключиться. Например, на компьютере с именем WIN81 нужно ввести «WIN81» (если вы выбрали вариант по умолчанию) или «WIN81\SQL2012» (если используется именованный экземпляр SQL2012).
3. Убедитесь, что в раскрывающемся списке выбран пункт **Проверка подлинности Windows**, и нажмите кнопку **Соединить**.
4. Установив соединение, нажмите клавишу F5, чтобы запустить скрипт. Сообщение об успешном выполнении скрипта будет выведено в нижней панели. БД **TSQL2012** должна появиться в списке доступных БД.
5. Закончив с этим, можете закрыть среду SSMS.

Чтобы создать и заполнить демонстрационную БД в рамках SQL Database, выполните следующие действия.

1. Откройте приложение SSMS, дважды щелкнув на файле **TSQL2012.sql** в «Проводнике». Перед вами появится диалоговое окно **Подключиться к компоненту Database Engine**.
2. Убедитесь, что в поле ввода **Имя сервера** указан именно тот сервер SQL Database, к которому вы хотите подключиться, например **myserver.database.windows.net**.
3. Убедитесь, что в поле **Проверка подлинности** выбран пункт **Проверка подлинности SQL Server**. Введите корректные логин и пароль. Нажмите кнопку **Параметры**.
4. В поле **Соединение с базой данных** во вкладке **Свойства соединения** введите **master**, затем щелкните на кнопке **Соединить**.
5. Скрипт разбит на несколько разделов. Пропустите раздел А (который предназначен для локальных экземпляров SQL Server) и следуйте инструкциям из раздела Б (для SQL Database). Самая важная инструкция содержит команду для создания БД **TSQL2012**.

```
CREATE DATABASE TSQL2012;
```

6. Щелкните правой кнопкой мыши на свободной области в окне с запросом и выберите пункт **Соединение ► Изменить соединение**. На экране появится диалоговое окно **Подключиться к компоненту Database Engine**. Укажите в качестве БД **TSQL2012** и нажмите кнопку **Соединить**. В списке **Доступные базы данных** должна появиться БД **TSQL2012**.

Как вариант, вы можете выбрать БД **TSQL2012** вручную, щелкнув на том же списке.

7. Выделите код начиная с раздела В (заголовок «Создание схем») и до самого конца файла. Нажмите **F5**, чтобы запустить скрипт. После успешного выполнения на нижней панели будет выведено соответствующее сообщение. Стоит отметить, что в условиях медленного сетевого соединения обработка кода может занять несколько минут.
8. В конце можете закрыть среду SSMS.

## Работа со средой SQL Server Management Studio

SQL Server Management Studio (SSMS) — это клиентская среда для написания и выполнения кода на языке T-SQL. Здесь вы не найдете исчерпывающего руководства по ее использованию. Главная цель этого раздела — помочь вам приобрести начальные навыки работы с данным инструментом.

Чтобы начать работу со средой SSMS, выполните следующие шаги.

1. Найдите и запустите программу SSMS в разделе **Microsoft SQL Server** в главном меню Windows.
2. Если вы делаете это впервые, советую откорректировать параметры запуска, чтобы они соответствовали вашим предпочтениям.
3. Если на экране появится диалоговое окно **Соединение с сервером**, нажмите кнопку **Отмена**.
4. Выберите пункт меню **Сервис ► Параметры**, чтобы открыть соответствующее окно. Перейдите в раздел **Среда ► Запуск** и выберите в списке **При запуске** пункт **Открыть браузер объектов и окно запроса**. Благодаря этому при каждом запуске среды SSMS будут открываться обозреватель объектов и новое окно запроса.
5. Обозреватель объектов — это инструмент для графического представления структуры объектов, который позволяет управлять работой SQL Server. Окно запроса предназначено для написания и выполнения кода на языке T-SQL. Можете просмотреть дерево настроек, чтобы лучше узнать о доступных вам вариантах, но не стоит рассчитывать на то, что вам сразу удастся в них разобраться. Позже, обретя некоторый опыт работы со средой SSMS, вы сможете к ним вернуться и внести все необходимые изменения.
6. Закончив знакомиться с диалоговым окном **Параметры**, нажмите кнопку **ОК**, чтобы подтвердить внесенные изменения.
7. Закройте среду SSMS и запустите ее снова, чтобы убедиться в том, что она автоматически открывает **Обозреватель объектов** и новое **Окно запроса**. Вы должны увидеть диалоговое окно **Соединение с сервером** (рис. П.12).
  - 7.1. В этом диалоговом окне необходимо указать информацию об экземпляре SQL Server, к которому вы хотите подключиться.
  - 7.2. В поле **Имя сервера** введите имя вашего сервера.

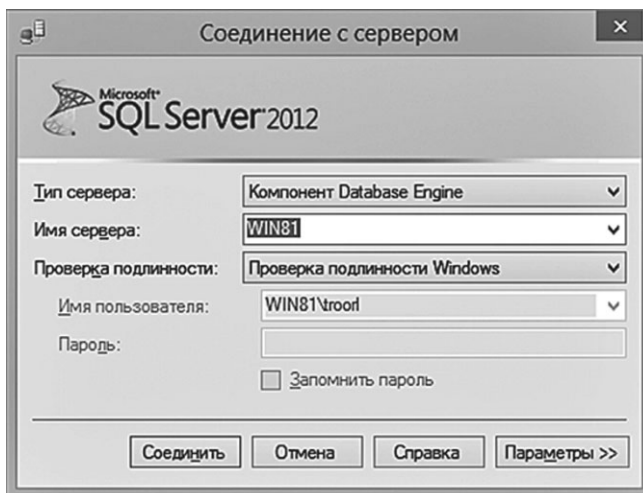


Рис. П.12. Диалоговое окно Соединение с сервером

7.3. Если вы имеете дело с локальным экземпляром SQL Server, убедитесь в том, что в списке **Проверка подлинности** выбран пункт **Проверка подлинности Windows**; если вы подключаетесь к SQL Database, выберите пункт **Проверка подлинности SQL Server** и укажите логин и пароль. Нажмите кнопку **Параметры**, перейдите на вкладку **Свойства соединения** и введите **TSQ2012** в поле **Соединение с базой данных**.

7.4. Нажмите кнопку **Соединить**. Перед вами должно появиться окно среды SSMS (рис. П.13).

Панель **Обозреватель объектов** находится слева, в центре размещено окно запроса, а справа вы можете видеть панель **Свойства** (которую лучше спрятать с помощью кнопки **Автоматически скрывать** в правом верхнем углу). Хотя эта книга посвящена разработке на языке T-SQL, а не управлению сервером БД, я настоятельно рекомендую вам тщательно исследовать содержимое панели **Обозреватель объектов** — пройдите по дереву элементов, раскрывая разные узлы. Это очень удобный инструмент для графического представления структуры и содержимого БД (рис. П.14).



## ПОДСКАЗКА

Вы можете перетаскивать элементы из панели **Обозреватель объектов** в окно запроса. Например, если перетащить таблицу, SQL Server автоматически добавит в код список всех ее столбцов, разделенных запятыми.

Код на языке T-SQL нужно вводить в окне запроса. Его выполнение происходит в контексте БД, к которой вы подключены. Локальный экземпляр SQL Server, в отличие от SQL Database, позволяет переключаться между разными БД. Для этого предусмотрен раскрывающийся список **Доступные базы данных** (рис. П.15).

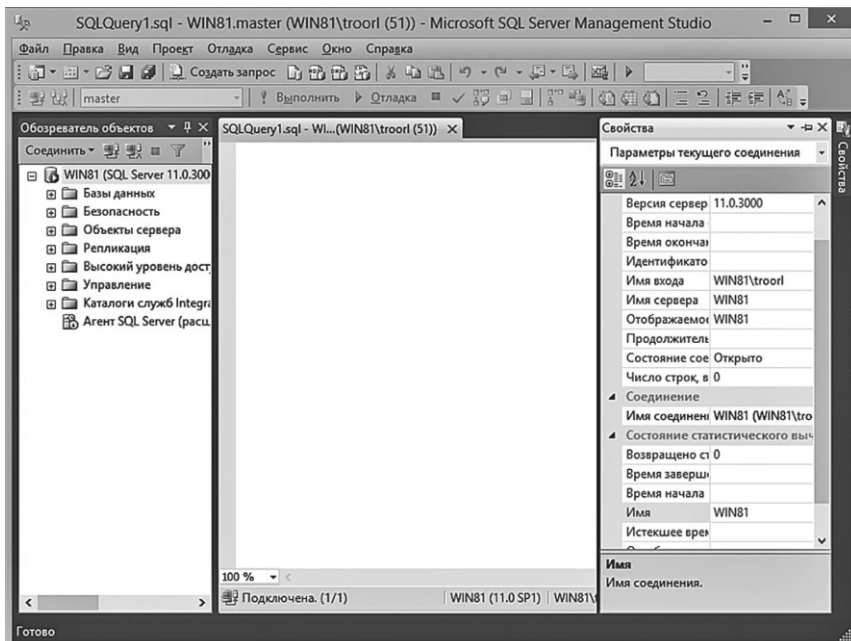


Рис. П.13. Начальный экран среды SSMS

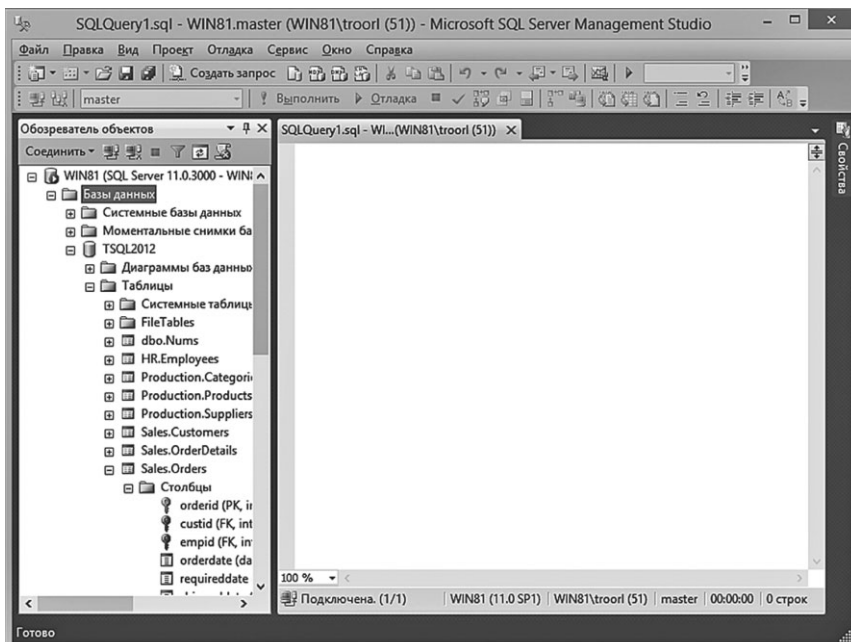


Рис. П.14. Обзор объектов

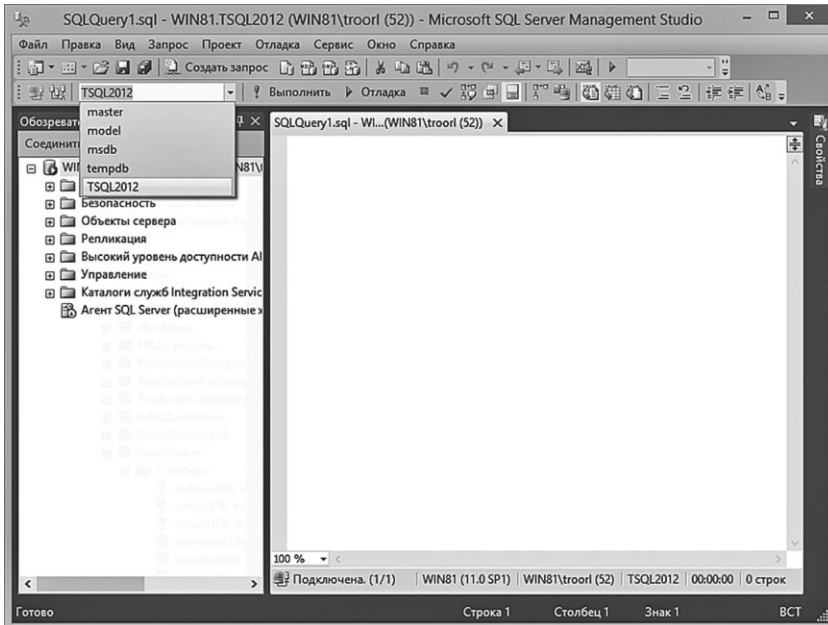


Рис. П.15. Раскрывающийся список Доступные базы данных

- 8) Убедитесь в том, что вы подключены к демонстрационной БД TSQ2012. Текущие сервер и БД можно изменить в любой момент; для этого достаточно щелкнуть правой кнопкой мыши в свободной области окна запроса и выбрать пункт **Соединение** ► **Изменить соединение**.
- 9) Теперь вы можете заняться разработкой на языке T-SQL. Наберите в окне запроса следующий код.

```
SELECT orderid, orderdate FROM Sales.Orders;
```

- 10) Чтобы его выполнить, нажмите клавишу F5. То же самое можно сделать с помощью кнопки **Выполнить** (значок с красным восклицательным знаком — не перепутайте его с зеленой стрелкой, которая запускает отладчик). Результат выполнения этого кода будет выведен на панели **Результаты** (рис. П.16).

С помощью раздела меню **Запрос** ► **Отправить результаты в** (или соответствующих кнопок на панели инструментов) легко изменять способ вывода результатов. Вам доступно три варианта: **В виде текста**, **В виде сетки** и **В файл**. Стоит отметить, что SQL Server по умолчанию выполняет только тот код, который был выделен в окне запроса (рис. П.17). Весь скрипт целиком выполняется только в случае отсутствия выделения.



#### ПОДСКАЗКА

Предварительно нажатая клавиша **Alt** позволяет выделять прямоугольную область кода, левая граница которой может начинаться в любом месте строки. Такое выделение тоже может выполняться отдельно от остального скрипта (рис. П.18).

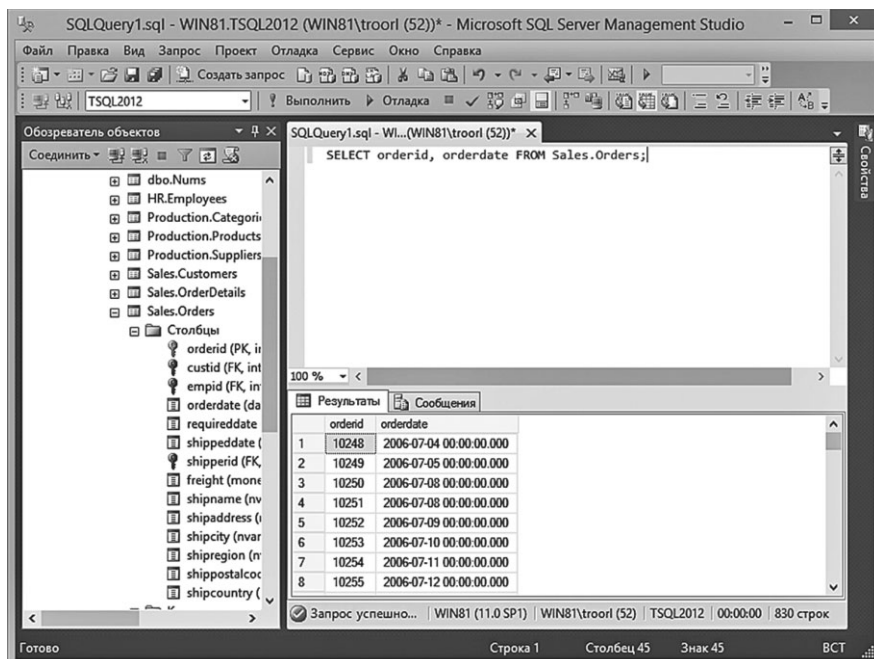


Рис. П.16. Результат выполнения первого запроса

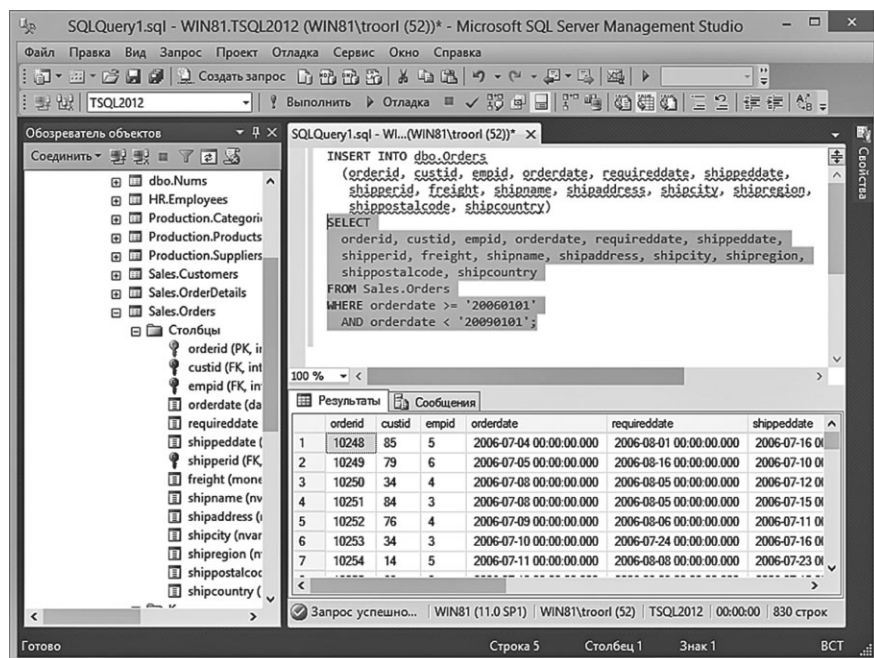


Рис. П.17. Выполнение выделенного участка кода

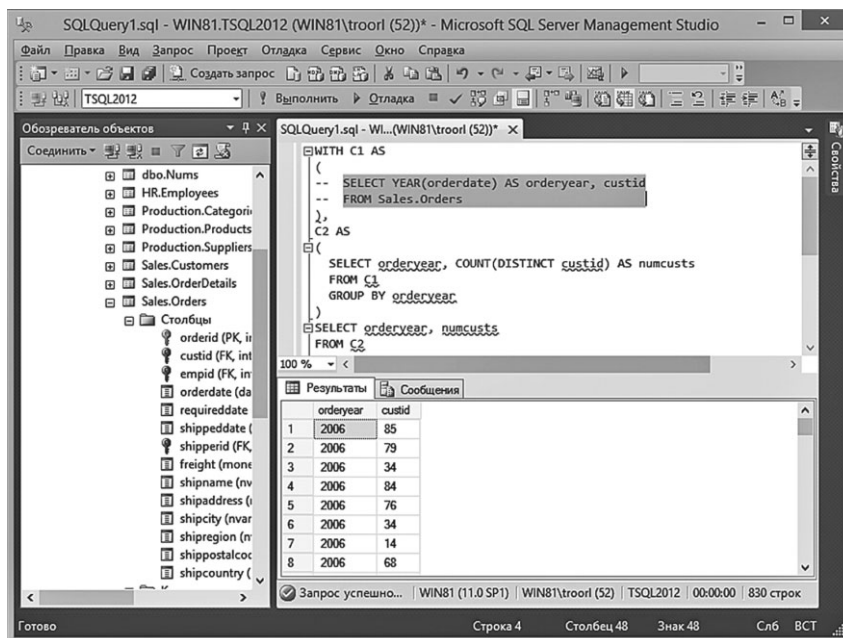


Рис. П.18. Выделение прямоугольного блока кода

Прежде чем дать вам возможность самостоятельно исследовать особенности среды SSMS, напомним, что весь исходный код для этой книги доступен по ссылке [eksmo.ru/smv/TSQLFundamentals.rar](http://eksmo.ru/smv/TSQLFundamentals.rar). Подробности ищите в предыдущем разделе. Допустим, вы уже загрузили и распаковали все файлы в локальный каталог; теперь можно открыть нужный вам скрипт, воспользовавшись пунктом меню **Файл ► Открыть ► Файл** или кнопкой **Открыть файл** на панели инструментов. То же самое можно сделать посредством «Проводника», дважды щелкнув на значке файла.

## Работа с электронной документацией

Компания Microsoft предоставляет исчерпывающую электронную документацию по SQL Server. Это самый полезный источник информации для разработки на языке T-SQL. Чтобы получить доступ к электронному справочнику, откройте главное меню Windows, перейдите в раздел **Microsoft SQL Server** и выберите пункт **Документация по SQL Server**. Если вы делаете это впервые, вам будет предложено выбрать источник информации по умолчанию — Интернет или локальное хранилище на вашем компьютере. Укажите тот вариант, который вам больше подходит; позже вы сможете изменить настройки в меню **Справка ► Управление параметрами справки** (или с помощью кнопки, которая находится в правом углу панели инструментов). Лично я решил хранить все справочные материалы по SQL Server 2012 на локальном диске.

Если вы последовали моему примеру, вам придется открыть **Диспетчер библиотек справки** и загрузить документацию из сети. Также стоит отметить, что содержимое электронного справочника не связано с выпусками пакетов обновлений для SQL



Server, поэтому обновления справочных материалов желательно проверять вручную, используя все тот же **Диспетчер библиотек справки**. Доступ к электронной документации по SQL Server 2012 можно получить, открыв страницу [msdn.microsoft.com/ru-ru/library/ms130214\(v=SQL.110\).aspx](http://msdn.microsoft.com/ru-ru/library/ms130214(v=SQL.110).aspx). Примеры в книге основаны на локальной копии электронного справочника, установленной на моем компьютере.

Работа с электронной документацией не требует особых навыков. Данный раздел был написан, чтобы вы узнали о самом факте существования справочных материалов и поняли, насколько они важны. Главное — понимать, что ответы на многие вопросы, связанные с SQL Server, можно найти самостоятельно — достаточно полистать электронную документацию.

Я остановлюсь на нескольких вариантах поиска полезной информации. Одна из наиболее часто используемых мною вкладок называется **Индекс** (рис. П.19).

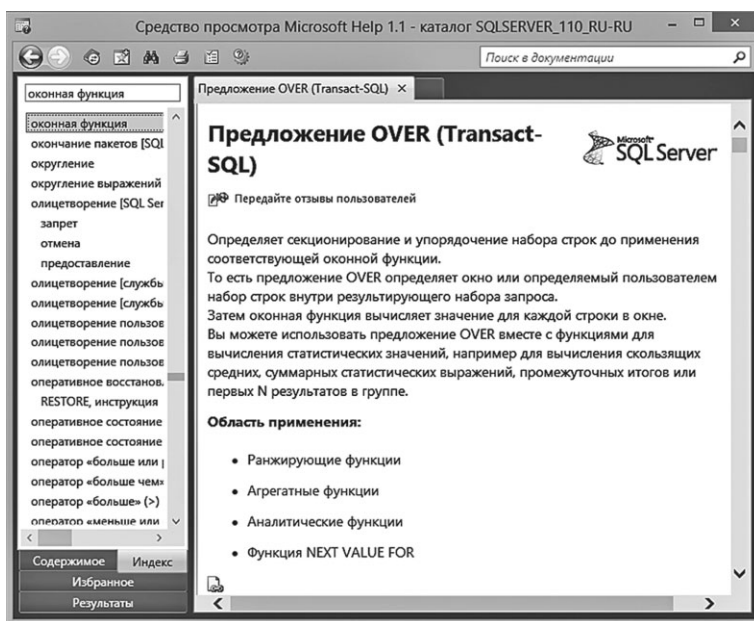


Рис. П.19. Вкладка **Индекс** электронной документации

Введите в поле **Указатель** то, что вы хотите найти (например, **оконная функция**). По мере ввода в отсортированном списке будут выделяться наиболее подходящие элементы. Вы можете ввести любую интересующую вас тему или ключевое слово языка T-SQL, если вам нужна информация о синтаксисе. Если вы хотите сохранить раздел, чтобы вернуться к нему позже, используйте кнопку **Добавить в избранное** на панели инструментов. Вы также можете синхронизировать текущий элемент с соответствующим разделом во вкладке **Содержимое**, нажав кнопку **Синхронизировать содержание**.

Если вы ищете не какой-то конкретный элемент, а целый раздел (например, «Новые возможности SQL Server 2012» или «Справочник по Transact-SQL»), вам следует обратить внимание на вкладку **Содержимое** (рис. П.20). На данной вкладке все материалы организованы в виде древовидного списка.

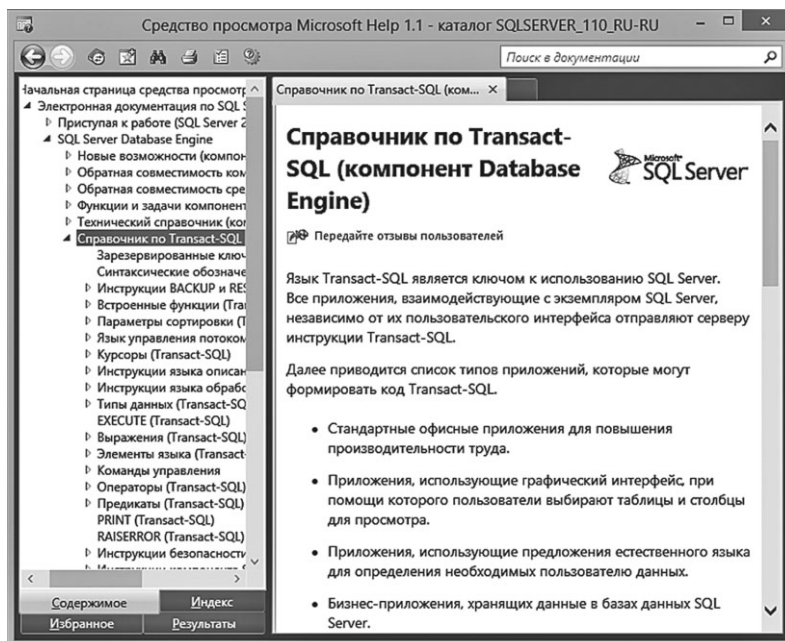


Рис. П.20. Вкладка **Содержимое** в электронной документации

Еще одной крайне полезной функцией является поиск по документации, доступ к которому легко получить с помощью поля ввода в правом верхнем углу (рис. П.21).

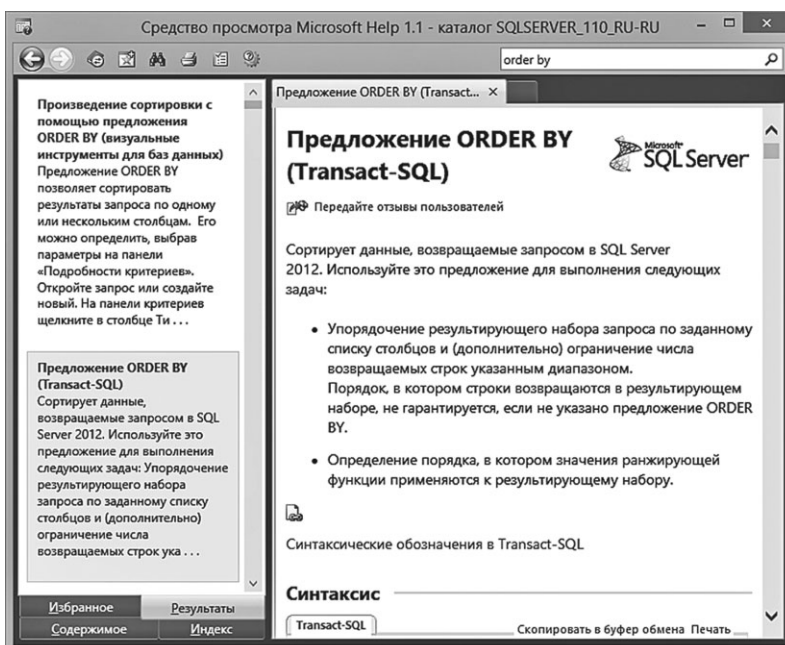


Рис. П.21. Поиск по электронной документации

Это более абстрактный вид поиска, чем тот, что используется во вкладке **Индекс**. Он чем-то напоминает интернет-системы, позволяющие осуществлять поиск статей по ключевым словам. Чтобы найти определенное слово в текущей статье, нажмите соответствующую кнопку на панели инструментов или воспользуйтесь комбинацией клавиш **Ctrl+F**.



#### **ПОДСКАЗКА**

---

Прежде чем вы приступите к самостоятельному изучению документации, позвольте дать вам последний совет. Если при написании кода внутри SQL Server Management Studio вам понадобится уточнить синтаксис того или иного элемента, установите текстовый курсор в нужное место и нажмите **Shift+F1** — на экране появится электронный справочник, открытый на странице с интересующим вас элементом (при условии, что информация о нем существует).

# АЛФАВИТНЫЙ УКАЗАТЕЛЬ

\* (звездочка) 56–57, 149–150  
+ (плюс), оператор 78–80  
\_ (подчеркивание), групповой символ 85  
% (процент), групповой символ 85  
\  
(обратный слеш) 32  
, (запятая) 53, 269  
; (точка с запятой) 38, 45, 275  
' (одинарные кавычки) 78  
" (двойные кавычки) 78  
() (круглые скобки) 66, 93, 165, 171  
{ } (фигурные скобки) 21

## A

ADO.NET 342  
ANSI SQL-89 112–113, 115–117  
ANSI SQL-92 112, 115, 117, 122

## B

BISM 27, 29

## D

DAX 27, 29  
DCL 19  
DDL 19–20, 354 *см. также* Хранилище данных  
DML 19–20, 251, 273, 287–289, 344, 354, 365–367  
DMX 27, 30  
DSA 27–28  
DW 27

## M

MDX 27, 29

## N

NULL, отметка 23–24, 37–41, 50, 70–74, 79, 94, 101, 122, 124–125, 142, 144, 199, 202, 223, 242, 345  
проблемы 152–155

## O

OLTP 27–28  
OSQL 342

## P

PowerPivot, система 27, 29

## S

SSMS *см.* SQL Server Management Studio  
SQL 18–19  
динамические возможности 358–361  
стандарты 19  
распределенный *см.* DSQL  
SQL Database 16, 31, 374–375  
SQL Server 18–19, 27–28, 38, 375–383  
архитектура 30–36  
разновидности 30–31  
справочник 46  
экземпляры 32, 375–386  
электронная документация 392–394  
SQL Server Management Studio 384, 386–391

## T

T-SQL 18, 20, 31, 34, 38

## W

Windows Azure SQL Database *см.* SQL Database

## A

Агрегат текущий 151–152  
Аргумент, использование 168–169, 172  
Атомарность 24–25, 298–299  
Атрибут 23

## Б

Базы данных 33–35  
Блокировка 301–310  
взаимная 324–327  
монопольная 301  
разделяемая 301–302  
ресурсы 302–304  
совместимость 302  
Булеан 240

## В

Витрина данных 28  
Выражение:  
CASE 67–70, 75

векторное 272  
табличное 165, 278–280  
— обобщенное *см.* OTB

## Д

### Данные:

добавление 251–265  
изменение 278–282  
изоляция 299  
надежность 299  
нормализация 24–26  
обновление 268–273  
согласованность 299  
слияние 273–277  
удаление 265–268

Декартово произведение 235

Диаграмма Венна 199

## Ж

Жизненный цикл данных 27–30

## З

Закон исключенного третьего 23

### Запрос:

вложенный 140, 152–157  
— автономный 140–146  
— коррелирующий 146–148  
— сложный 150–152  
внешний 140  
внутренний 140  
недетерминированный 61  
порядок обработки 44–45  
продвинутый 217–244

### Значение

@params, 360  
@stmt 360  
отсутствующее неприменимое 23  
— применимое 23  
скалярное 63

## И

Индекс 35, 39, 47, 92, 102, 107, 255, 326, 349, 359

### Инструкция

ALTER TABLE 39, 96, 259, 303  
BEGIN TRAN 298  
COLLATE 77  
COMMIT TRAN 298–301  
CREATE TABLE 37, 39, 57, 255, 259, 366, 368  
CUBE 240

DECLARE 339, 356  
DISTINCT 145, 198–201, 205, 210, 222–223  
ELSE 68–69, 230, 346  
FROM 44–46, 64, 111, 118, 121, 124, 128, 165, 170, 185, 237, 253, 255, 267, 362  
GROUP BY 44, 48–51, 64, 73, 167–168, 209, 223, 230–231, 239–240  
GROUPING 239  
GROUPING SETS 239–240  
HAVING 44, 48, 51–52, 54, 64, 68, 70, 208  
IF 37, 348  
IF...ELSE 345–346  
INTO 255, 283, 285  
ON 115–116, 119, 122–124, 128, 267  
ORDER BY 44, 48, 57–59, 61–64, 68, 73, 166, 177–180, 198, 203, 208–209, 218–219, 224–225, 280–282, 348  
OUTPUT 269, 283–288  
OVER 63–64, 217–218, 221, 263  
PARTITION BY 63, 203, 219, 226  
ROLLBACK TRAN 298, 301, 366  
ROLLUP 239–241  
SELECT 20, 36, 43–45, 48, 52–57, 64, 74, 141, 149, 165–168, 177, 182, 219, 222–223, 230, 269, 272, 278–279, 282, 304, 326, 365  
SET 78, 269–271, 279, 297, 339–341  
THEN 68–69  
USE 37, 44  
USING 275–276  
VALUES 214, 235, 252–253–267  
WHEN 68–69, 75  
WHEN MATCHED 275–277, 296–297  
WHEN NOT MATCHED 275–276, 297  
WHEN NOT MATCHED BY SOURCE 276–277  
WHERE 45–48, 51, 54, 64–65, 68, 70, 74–75, 116, 124, 127, 167, 208, 219, 266–267, 269  
WHILE 346–348  
WITH 171–172, 180

## К

### Ключ:

внешний 24, 40–41, 118–119  
первичный 39, 118–119  
потенциальный 24  
резервный 24

### Команда:

BULK INSERT 251, 256  
CREATE DATABASE 368  
CREATE SEQUENCE 260–261

DELETE 20, 177, 251, 266–268, 271, 273,  
278, 280, 284, 287, 318, 366  
EXEC 359  
GO 342, 344–345  
INSERT 114, 177, 214, 258–259, 273, 276,  
278, 280, 283–284, 289, 316, 366  
INSERT EXEC 251, 253–254  
INSERT SELECT 251, 253–255  
INSERT VALUES 251–253  
MERGE 20, 251, 273–277, 287–288, 297, 366  
SELECT INTO 251, 254–255  
TRUNCATE 266–267  
UPDATE 70, 177, 251, 262, 269–273, 276,  
278–283, 286–287, 289, 318, 366

Конструктор табличных значений 235, 252–253, 272

Курсор 58, 60, 348–352

## Л

Литералы строковые 88–91

Логика троичная 23

## М

Метаданные, извлечение 100–103

Множество 20–22, 55, 199–200, 340, 348–349

бесконечное 21

пересечение 201–202

разность 204–206

Мультимножество 20, 55, 58, 198, 199–200,  
201–204, 206, 289, 348

## Н

Набор

группирующий 238–244

операторы для работы 198–210

## О

Облачные сервисы 31

Обновление потерянное 315–316

Обработка ошибок 369–373

Объект 35

последовательности 260–265

программируемый 339–373

Ограничения 24

CHECK 41

DEFAULT 41–42

UNIQUE 39

ключа внешнего 40–41

— первичного 39

Операторы:

AND 65–67, 277

APPLY 111, 185–188, 308

EXCEPT 198, 204–206

EXCEPT ALL 206

INTERSECT 201–202

INTERSECT ALL 202–204

JOIN 111, 113, 122, 124, 231, 267–268, 270–272

NOT 66, 144, 153

OR 65–67

PIVOT 111, 231–233, 358, 360–361

UNION 199, 200–201

UNION ALL 199–200

UNPIVOT 111, 237

арифметические 66

инвертирования 66

логические 66

объединения строк 78–80

работы с наборами 198–210

сравнения 65

табличные 111

Операции:

одновременные 74–75

сравнения 76–78

ОТВ 165, 171–176

Отношение 22

ссылочное 24

## П

Пакет 341–345

анализ 342

разрешение имен 343–344

Параметр:

ASC 58

CHECK OPTION 182–183

CYCLE 261

DATEFORMAT 88

DESC 58

DISTINCT 50, 56, 59–60, 64

ENCRYPTION 180–181

SCHEMABINDING 181–182

WITH TIES 62–63

Переменная 339–341

область видимости 342–343

табличная 356–357

Предикат 21, 23, 65

BETWEEN 65

EXISTS 148–150

IN 65, 142–144

LIKE 65, 84–86

Представления 165, 176–183

информационная схема 102

каталог 101–102

параметры 180–183

Программирование, стили 19

Программно-аппаратный комплекс 30

Процедура 362–369

sp\_columns 102

sp\_executesql 359–360

sp\_help 102

sp\_helptext 181

sp\_helpconstraint 103

sp\_refreshsqlmodule 177

sp\_refreshview 177

sp\_sequence\_get\_range 264

sp\_tables 102

системная хранимая 102–103, 364–365

## Р

Разворачивание данных 217

данных 228–230

— с помощью оператора PIVOT 231–233

— стандартными средствами 230–231

отмена 233–234

— с помощью оператора UNPIVOT 237

— стандартными средствами 235–236

Реляционная логическая модель 18–19, 21–22, 27

## С

Самосоединение перекрестное 113–114

Свойство:

collation 34

identity 256–260, 264

Секционирование окон 218–219, 221, 224–225, 227

Символ:

групповой 85–86

экранирования 86

Система управления базами данных 18, 22

Соединение:

внешнее 122–127, 130–131

внутреннее 115–117

естественное 119

множественное 121–122

— внешнее 128–129

не-экви- 119–121

перекрестное 111–115

составное 118–119

тета- 119

условие 115

экви- 119

Ссылки множественные 170–171, 173–174

Столбец 21, 167–168, 171–172

Строки 21, 78–80

Структура:

звезды 28

снежинки 28

Схема 35–36, 102

## Т

Таблица 21–22, 27

вложенная 169–170

временная 352–357

производные 165–171

создание 37–38

числовые 114–115

Тип:

INT 23

SQL\_VARIANT 355

SYSNAME 355

VARCHAR 38

данных 23, 75–76

— монолитный 87

дата и время 87, 90–92

приоритет 66–67, 88

табличный 357–358

Транзакция 34–35, 298–301

Триггер 366–369

## У

Уровень изоляции 311–312, 324

READ COMMITTED 301, 304, 311, 313–314, 324

READ COMMITTED SNAPSHOT 322–324

READ UNCOMMITTED 311–313, 324

REPEATABLE READ 314–316, 324

SERIALIZABLE 316–317, 324

SNAPSHOT 318–320, 324

обнаружение конфликтов 320–323

управление версиями строк 317–324

## Ф

Фильтр:

OFFST-FETCH 62–63, 186–187, 280–282

TOP 59–62, 187, 280–282

Фильтрация дат 92–94

ФТЗ встроенные 165, 184–188

Функции:

\$action 287

@@identity 257–258

CAST 94, 148  
CHARINDEX 81  
CHOOSE 69–70  
COALESCE 69–70, 79  
COLUMNPROPERTY 103  
CONCAT 78, 80  
CONVERT 90, 94–95  
COUNT 130–131  
DATABASEPROPERTYEX 103  
DATALENGTH 81  
DATEADD 96–97, 126  
DATEDIFF 97–98, 125  
DATENAME 99  
DATEPART 98  
DAY 98–99  
DB\_NAME 306–307  
DENSE\_RANK 220–221  
EOMONTH 100  
ERROR\_LINE 370  
ERROR\_MESSAGE 370  
ERROR\_NUMBER 370–371  
ERROR\_PROCEDURE 370  
ERROR\_SEVERITY 370  
ERROR\_STATE 370  
EVENTDATA 368–369  
FIRST\_VALUE 223–225  
FORMAT 84  
GROUPING 242–243  
GROUPING\_ID 241–244  
IDENT CURRENT 258–259, 260  
IF 69  
ISDATE 99  
ISNULL 69  
LAG 223–225, 357  
LAST\_VALUE 223–225  
LEAD 223–225  
LEFT 80, 85  
LEN 80–81  
LOWER 83–84  
LTRIM 84  
MONTH 98–99  
OBJECT\_ID 37, 103  
OBJECT\_NAME 307  
NEXT VALUE FOR 261–263, 284  
NTILE 220–221  
OBJECT\_DEFINITION 180–181  
OBJECTPROPERTY 103  
PARSE 91, 94–95

PATINDEX 81, 84  
RAND 362  
RANK 220–221  
REPLACE 81–82  
REPLICATE 82–83  
RIGHT 80, 82  
ROW\_NUMBER 63–64, 194, 203, 206, 210, 220–223, 279–280  
RTRIM 84  
SCHEMA\_NAME 101  
SCOPE\_IDENTITY 257–258, 283  
SERVERPROPERTY 103  
STUFF 83  
SUBSTRING 80, 85  
SWITCHOFFSET 96  
TODATETIMEOFFSET 96  
TRY\_CAST 94  
TRY\_CONVERT 94  
TRY\_PARSE 94  
TYPE\_NAME 101  
UPPER 83–84  
YEAR 98–99  
агрегатные 49–51  
даты и времени 93–100  
оконные 63–64, 217–219  
— агрегатные 225–228  
— ранжирующие 220–223  
— со смещением 223–225  
пользовательские 362–363  
системные хранимые 102–103

**X**

Хранилище данных 28

**Ц**

Целостность данных 24, 38–42

**Ч**

Чтение:

грязное 312–313  
неповторяющееся 314  
повторяющееся 314–316  
фантомное 316

**Э**

Элемент:

закрепленный 174–175  
порядок размещения 21  
рекурсивный 174–176



# Microsoft SQL Server 2012

## Основы T-SQL

**Изучите язык T-SQL и начните писать идеальные запросы**

Освойте основы языка Transact-SQL и начните разрабатывать собственный код для извлечения и изменения данных Microsoft SQL Server 2012. Вы изучите концепции, лежащие в основе программирования и написания запросов для Microsoft SQL Server 2012, и затем сможете применить свои знания при решении упражнений, которые приводятся в конце каждой главы. Разобравшись с логикой языка T-SQL, вы быстро научитесь писать эффективный код вне зависимости от того, программист вы или администратор баз данных.

### Вы узнаете, как:

- работать с уникальными программными инструкциями языка T-SQL
- создавать таблицы и обеспечивать целостность данных
- обращаться одновременно к нескольким таблицам, используя соединения и вложенные запросы
- упростить свой код и улучшить его сопровождаемость с помощью табличных выражений
- реализовать разные методики добавления, обновления, удаления и слияния данных
- использовать продвинутые средства, такие как оконные функции, разворачивание данных и группирующие наборы
- регулировать согласованность данных с помощью уровней изоляции, избегая обычного и взаимного блокирования
- вывести свои запросы на новый уровень за счет применения программируемых объектов



Примеры кода доступны в Интернете. Загрузите их по адресу:

[eksmo.ru/smv/TSQLFundamentals.rar](http://eksmo.ru/smv/TSQLFundamentals.rar)

Системные требования приведены во введении



### Об авторе:

Ицик Бен-Ган – ведущий специалист в области SQL Server с 1999 года, один из основателей проекта SolidQ, в рамках которого он занимается преподаванием и консультированием по всему миру, освещая такие темы, как программирование на языке T-SQL и оптимизация запросов. Регулярно публикуется в журналах SQL Server Pro и MSDN Magazine, выступает на крупных отраслевых конференциях, включая Microsoft Tech-Ed, DevTeach, PASS и SQL Server Connections.

*Прекрасная книга для начинающих. Масса полезной информации, примеров кода, диаграмм и советов. Отличный способ освоить T-SQL.*

Иэн Стирк,  
программист и архитектор

**Базы данных/Microsoft/SQL Server**

**Microsoft**



ЭКСМО

ISBN 978-5-699-73617-1



9 785699 736171 >