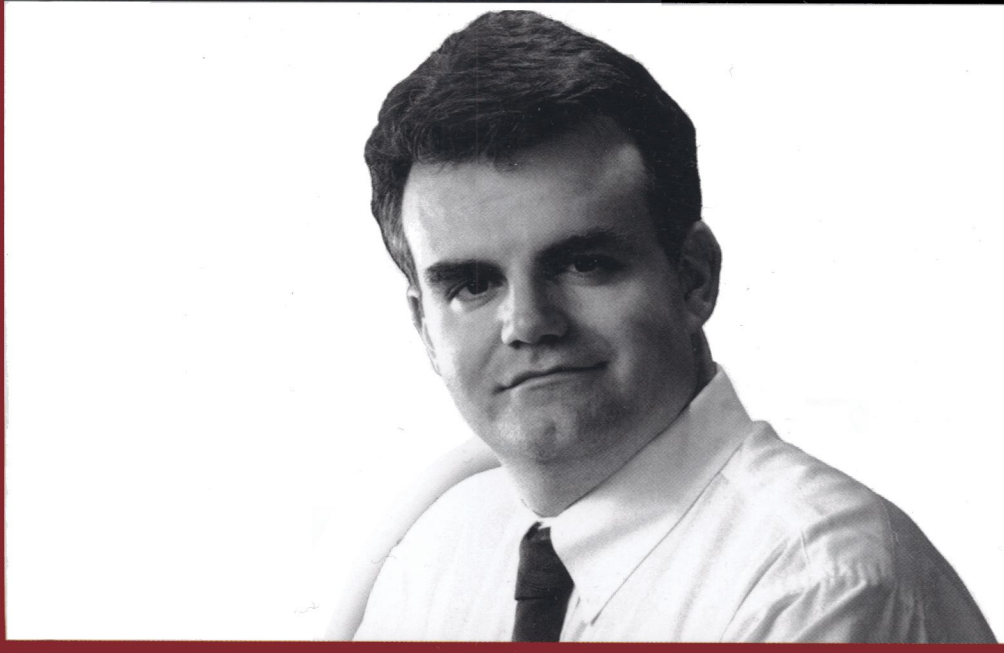
Программистам от программистов



Программирование баз данных Microsoft SQL Server™ 2005

Базовый курс

Роберт Виейра

Обновления, исходный код и техническая поддержка предоставляются компанией Wrox на узле **www.wrox.com**

www.dialektika.com

Beginning SQL Server™ 2005 programming

Robert Vieira

Wiley Publishing, Inc.

Программирование баз данных Microsoft SQL Server™ 2005

Базовый курс

Роберт Виейра

"Диалектика"
Москва • Санкт-Петербург • Киев
2007

ББК 32.973.26-018.2.75
В42
УДК 681.3.07

Компьютерное издательство "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *К.А. Птицына*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Виейра, Роберт.

В42 Программирование баз данных Microsoft SQL Server 2005. Базовый курс. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2007. — 832 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1202-2 (рус.)

В настоящей книге SQL Server 2005 рассматривается главным образом с точки зрения разработчика. В ней описаны основные особенности этой версии СУБД, позволяющие значительно улучшить совместимость компонентов и расширить набор средств, обеспечивающих взаимодействие с языком XML, инфраструктурой .NET, определяемыми пользователем типами данных, а также со многими другими дополнительными службами. Книга целиком посвящена изложению основных сведений о средствах разработки, необходимых каждому разработчику независимо от уровня его подготовки, и содержит описание современной версии рассматриваемого программного продукта, но включает все необходимые сведения о проблемах обеспечения обратной совместимости, которые могут повлиять на выбор наиболее подходящих способов проектирования и написания кода.

Книга предназначена для разработчиков всех уровней, работающих с базами данных Microsoft.

ББК 32.973.26-018.2.75

ISBN 978-5-8459-1202-2 (рус.)

ISBN 0-7645-8433-2 (англ.)

Оглавление

Содержание

От всего сердца посвящаю эту книгу моим дочерям, Эшли и Эдди, которые терпеливо переносили мои "исчезновения" в домашнем офисе на протяжении тех месяцев, когда я писал эту книгу. Они дали мне силы для успешного завершения поставленной задачи, и моя любовь к ним не знает границ. Я просил бы издательство Wiley только об одном — чтобы мне позволили вынести на обложку не мой скромный портрет, а прекрасные лица этих двух женщин.

Об авторе

Роберт Виейра (Robert Vieira) заразился "компьютерной лихорадкой" в 1978 году и с тех пор не расстается с мыслью, что тогда в его жизнь вошло нечто "поистине восхитительное". В 1980 году началось его более полное погружение в мир вычислительной техники. Ему приходилось уделять часть времени созданию и восстановлению вычислительных комплексов, а в другое время заниматься программированием на языке Basic и на языках ассемблера для вычислительных систем Z80 и 6502. В 1983 году Роб начал готовиться к сдаче экзаменов на степень бакалавра по компьютеризированным информационным системам, но пришел к выводу, что профессиональная деятельность в среде мэйнфреймов слишком далека от того, чем он хотел бы заниматься, и оставил учебу в 1985 году, чтобы найти область деятельности, в большей степени соответствующую его интересам. Позднее в том же году он стал настоящим "рабом персонального компьютера" и вступил на долгий путь профессионального роста в области программирования на языках баз данных, от dBase до SQL Server. Роб защитил степень бакалавра делового администрирования в 1990 году и с тех пор, как правило, работает на должностях, которые позволяют ему успешно применять свои знания в области управления коммерческой деятельностью и компьютерных наук. Кроме степени бакалавра, Роб прошел аттестацию в качестве профессионального бухгалтера-аналитика, а также является обладателем "Сертификата разработчика решений Microsoft" (MCSD), "Сертификата преподавателя Microsoft" (MCT) и "Сертификата администратора баз данных Microsoft" (MCDBA).

В настоящее время Роб занимает должность разработчика структуры системы программного обеспечения в компании WebTrends, г. Портленд, штат Орегон.

Роб проживает со своими дочерьми Эшли и Адрианной в Ванкувере, штат Вашингтон.

Благодарности

Прошло пять лет с тех пор, как я написал свою первую книгу, посвященную СУБД SQL Server. За это время моя жизнь коренным образом изменилась. Я воспринял разностороннее влияние очень многих людей, но, как всегда, буквально не имею возможности лично поблагодарить всех тех, к кому испытываю глубокую признательность.

Начну выражения благодарности со своих детей, которые непостижимым образом продолжают оставаться добрыми и нежными даже при таком папе, всегда пытающемся навязать свое мнение. Хотя моя младшая уже несколько раз спрашивала, когда я собираюсь "покончить с этой книгой" (поскольку не рада тому, что у нас почти не остается времени, чтобы поиграть друг с другом), она проявляла необычайное терпение в течение всего времени написания книги. А моя старшая продолжает поражать меня зрелостью своих суждений и пониманием всего того, что требуется для подготовки книги (а также, безусловно, того, как это отразится на ее обучении в колледже!). Безусловно, я прежде всего обязан сказать спасибо своим детям.

Я очень благодарен своим читателям. Они присылают мне письма и всегда сообщают, сумел ли я им в чем-то помочь. В этом состоит одна из причин, по которым я в очередной раз мобилизую все свои силы, чтобы написать еще одну книгу. Непрекращающаяся поддержка моих стараний в области создания серии книг для профессионалов просто удивительна. По-видимому, мне удалось найти возможность предоставить читателям то, в чем они нуждаются, и я этому чрезвычайно рад. Остается надеяться, что благодаря этому для моих читателей хотя бы немного упростится освоение нюансов работы с СУБД SQL Server, а их производственная деятельность станет намного более успешной.

Я также хочу выразить особую признательность многим людям, с которыми я сталкивался в прошлом и настоящем. Некоторые из них работали еще в старом коллективе Wrox Press, и я давно потерял с ними контакт, но не забываю, как много они для меня сделали в период моего становления в качестве писателя. С другими представителями издательства мне довелось познакомиться сравнительно недавно, но и они сумели показать себя наилучшим образом. Иногда им для этого достаточно было лишь немного запастись терпением. Назову всех поименно.

- Кейт Холл (Kate Hall). Кейт участвовала в подготовке двух моих первых книг. Иногда казалось, что ко времени окончания работы над очередной книгой она готова была меня уничтожить, но так или иначе процесс редактирования каждый раз завершался достижением наилучшего результата из всех возможных. Я давно потерял связь с Кейт, но она всегда останется для меня тем человеком, который действительно помог мне сделать карьеру писателя. Наверное, я всегда буду оставлять это место среди первых "профессиональных" посвящений для вас, Кейт, где бы вы ни были. Надеюсь, что и на новой работе вы продолжаете проявлять себя так же блестяще.
- Адаоби Оби Талтон (Adaobi Obi Tulton). Адаоби была рядом со мной в течение еще одного напряженного года моей жизни и всегда каким-то образом умудрялась добиваться выполнения графика подготовки книги. Если бы я когда-либо стал богатым, то обязательно пригласил бы Адаоби к себе в качестве духовного наставника. Направляя значительные усилия на обеспечение своевременного осуществления намеченных планов, она вместе с тем создает вокруг себя незримую атмосферу спокойствия и умиротворения во всем том, чем занимается. Я сам хотел бы этому научиться.
- Доминик Шейкшафт (Dominic Shakeshaft). Именно Доминик направил меня на писательскую стезю (и теперь я уже не могу с нее сойти, несмотря на все трудности и бессонные ночи...).
- Кэтрин Александер (Catherine Alexander). При подготовке моей первой книги Кэтрин в основном только помогала Кейт, но вторая моя книга главным образом была поручена ей. Я обязан Кэтрин не меньше, чем Кейт, поскольку она оказала существенное влияние на содержание моих двух первых книг и обеспечила их успех.
- Джон Мюллер (John Mueller). Джону было поручено самое сложное задание — выявление допущенных мною ошибок. Я сам когда-то занимался техническим редактированием и знаю, насколько сложно проверить все до последней мелочи, заметив при этом даже малейшие пропуски и самые незначительные ошибки. Но еще тяжелее бывает, когда приходится, взяв на себя обязанности по исправлению результатов чужого труда, принимать решение, следует ли порекомендовать писателю внести какие-либо исправления или оставить все так, как есть. Джон справляется и с той и с другой работой с поразительным успехом.

В подготовке настоящей книги к выпуску участвовало гораздо меньше сотрудников издательства по сравнению с предыдущими книгами, но в связи с тем, что данная книга создавалась так долго и к издательскому процессу имело отношение такое количество людей, я, безусловно, не сумею выразить свою благодарность всем, кто этого заслуживает. Тем, кто не указан в этом списке, я приношу свои самые скромные извинения и еще раз подчеркиваю, что глубоко ценю вашу помощь. Несмотря на сказанное, хочу отдельно назвать имена тех людей, которым я особенно признателен (некоторые из них — мои старые друзья): Пол Терли (Paul Turley), Грэг Бимер (Greg Beamer), Итцик Бен-Ган (Itzik Ben-Gan), Кейлен Делане (Kalen Delaney), Фернандо Геррего (Fernando Guerrero), Джерт Дрейперс (Gert Drapers) и Ричард Вэймайр (Richard Waymire).

Введение

Кажется, это было совсем недавно, но какие с тех пор произошли существенные изменения! В то время, когда автор занимался подготовкой к печати книги *Professional SQL Server 7.0 Programming* в начале 1999 года, для разработки применялись во многом иные инструментальные средства, чем в настоящее время. В тот период времени не было ни малейшего представления о том, что когда-либо появится инфраструктура .NET, а доминирующее положение в качестве среды разработки занимала программа Visual Studio 98. Значительная доля рынка принадлежала языку Java, а такие мощные альтернативные инструментальные средства разработки, как Delphi, были намного более конкурентоспособными, чем сегодня. Стремительно увеличивалось количество Интернет-компаний (так называемых "дот-комов"), а сфера применения систем управления базами данных (СУБД), таких как SQL Server, постоянно росла.

В связи с широким распространением баз данных возникла следующая проблема. Количество книг, посвященных описанию СУБД SQL Server, было достаточно велико, но все эти книги были предназначены для администраторов базы данных, поэтому подавляющая часть содержимого этих книг была посвящена изложению такой тематики, которая не имеет не малейшего отношения к работе разработчика среднего уровня. Нужно было что-то срочно предпринять, а поскольку мой редактор в издательстве и я были хорошо знакомы с современным состоянием дел, то пришли к выводу, что сможем удовлетворить нереализованную потребность в книге по языку SQL, которая должна быть предназначена специально для разработчиков.

Таким образом, была выпущена книга *Professional SQL Server 7.0 Programming*, которая должна была предоставить всем разработчикам все, что им требуется. Эта книга была задумана как своего рода энциклопедия, в которой прежде всего приведены начальные сведения, а затем в ходе дальнейшего изложения раскрывается все до малейших подробностей. В результате получилась очень и очень большая книга, которая оказалась нужной для весьма многих людей.

С тех пор были выпущены две основные версии СУБД SQL Server, последней из которых стала версия SQL Server 2005. Но после того как я с представителями издательства приступил к планированию цикла книг, посвященного этой версии, мы поняли, что снова столкнулись с проблемой — объем намеченной книги должен был стать слишком большим. В версии SQL Server 2005 было реализовано так много новых средств, что возникла ситуация, когда было просто невозможно вместить такой большой объем информации в одну книгу. В связи с этим было принято решение вместо одной книги старой серии "для профессионалов" выпускать две книги. Одна из них относится к серии "базовый курс", а вторая сохраняет название серии "для профессионалов", но является более целенаправленной. Настоящая книга явилась результатом реализации первой части этого замысла.

Я надеюсь, что в данной книге мне удалось описать все основные компоненты СУБД SQL Server с таким же успехом, которым славились предыдущие книги по программированию SQL Server для профессионалов. Закончив чтение этой книги, читатель сможет стать весьма хорошо подготовленным программистом для SQL Server 2005 и в случае необходимости перейти к изучению более сложных книг серии для профессионалов.

Для кого предназначена эта книга

Остается только пожалеть, что я вместе с издательством не смог подобрать другое выражение, чтобы обозначить направленность этой книги, кроме как "базовый курс". Не поймите меня неправильно; эта книга действительно предназначена для тех, кто только знакомится с СУБД SQL Server. Но замысел настоящей книги таков, что она будет служить вам еще очень долго, после того как первые шаги останутся уже далеко позади. Информация, приведенная в этой книге, действительно является необходимой для начинающих разработчиков, просто объем ее слишком велик для того, чтобы ее можно было постоянно держать в памяти. Поэтому материал этой книги изложен в такой форме, которая позволяет надежно усвоить необходимые сведения, а затем снова возвращаться к этому описанию, став более подготовленным и даже очень квалифицированным пользователем.

Рекомендуем начинающему пользователю приступать к изучению данной книги с самых первых ее строк. Изложение в этой книге построено так, что почти все в ней относится к такой категории сведений, которые действительно необходимо знать. Возможно, за исключением глав, посвященных описанию языка XML, а также служб Reporting Services и Integration Services, каждый раздел настоящей книги содержит важные сведения, изучение которых позволяет получить полное представление о том, какой подход к решению задач программирования для SQL Server является наиболее приемлемым.

Если читатель уже имеет определенную подготовку, то может на первых порах пропустить главы вплоть до 7 или 8. Безусловно, следует тем не менее порекомендовать ознакомиться и с предыдущими главами, чтобы восполнить пробелы в своих знаниях или получить более полное представление о рассматриваемой теме. Возможно, пропустив начальные главы, вы также сумеете успешно усваивать дальнейший материал, хотя и с некоторыми дополнительными трудностями.

Квалифицированные пользователи могут обращаться к этой книге как к превосходному источнику справочной информации. Возможно также, что они захотят более подробно ознакомиться с главой 12 и следующими главами. Определенный интерес практически для любого разработчика представляют фактически все сведения, которые изложены в последних главах (где речь идет о новых средствах отладки, транзакциях, языке XML, службах Reporting Services и многом другом).

Информация, представленная в книге

Прежде всего следует еще раз подчеркнуть, что в настоящей книге программное обеспечение SQL Server 2005 рассматривается главным образом с точки зрения разработчика.

SQL Server 2005 — это новейшая версия одной из систем управления базами данных, достигшая того непревзойденного уровня развития, к которому она постепенно приближалась в течение двух десятилетий. Современная версия явилась результатом коренной переработки, которой подвергся этот программный продукт, начиная с версии 7.0. Но в программном обеспечении SQL Server 2005 удалось значительно улучшить совместимость компонентов и расширить набор средств, обеспечивающих взаимодействие с языком XML, инфраструктурой .NET, определяемыми пользователем типами данных, а также со многими другими дополнительными службами.

Настоящая книга целиком посвящена изложению основных сведений о средствах разработки, необходимых каждому разработчику независимо от уровня его подготовки. Излагаемый материал в основном содержит описание версии SQL Server 2005 данного программного продукта, но включает все необходимые сведения о проблемах обеспечения обратной совместимости, которые могут повлиять на выбор наиболее подходящих способов проектирования и написания кода.

Общая структура книги

Настоящая книга подготовлена так, что переход к изложению все более и более сложных вопросов происходит постепенно, по мере ее дальнейшего изучения. Однако автор с самого начала исходит из предположения, что читатель уже является достаточно опытным разработчиком; под этим не обязательно подразумевается опыт работы с базами данных. Чтобы иметь возможность успешно осваивать материал, изложенный в данной книге, читатель уже должен быть знакомым с основами программирования, такими как переменные, типы данных и принципы процедурного программирования. Тем не менее читатель может не иметь ни малейшего представления о языке SQL (хотя вполне можно предположить, что число таких читателей будет весьма невелико).

Настоящая книга в основном предназначена для разработчиков. Это означает, что в целях сокращения и упрощения некоторые темы, в большей степени относящиеся к сфере деятельности администратора базы данных, а не разработчика, излагаются очень кратко или полностью игнорируются. Тем не менее не остаются без внимания некоторые вопросы администрирования, которые имеют важное значение для разработчика или должны учитываться в процессе разработки. Кроме того, в главе 19 приведен краткий обзор нескольких тем, касающихся администрирования.

При подготовке настоящей книги были предприняты значительные усилия с целью того, чтобы изложение в ней не было привязано к какому-то конкретному языку, относящемуся к сфере разработки клиентских программ. Специфика, связанная с применением VB, C#, C++, Java и других языков, в основном игнорировалась (поскольку книга в большей степени посвящена созданию серверных компонентов приложений), а если речь идет о каких-либо языках программирования, то ни одному из них не отдается предпочтение.

Порядок изложения учебного материала выбран так, что вначале приведены сведения об основных объектах, применяемых в языке SQL, а затем происходит переход к описанию наиболее широко используемых запросов и соединений. С этого момента начинают рассматриваться вопросы создания объектов в базе данных, а также затрагиваются темы, имеющие важное значение для физического проектирования. Иными словами, после изложения основ начинается описание таких важных средств программирования SQL Server, как сценарии, хранимые процедуры, пользовательские функции и триггеры. После этого приведено описание некоторых относительно менее важных средств СУБД SQL Server. Книга завершается описанием задач администрирования, осуществление которых позволяет обеспечить надежную и бесперебойную работу создаваемой базы данных.

Компоненты, необходимые для эффективного использования книги

Для того чтобы действительно получить какую-то практическую пользу от изучения настоящей книги, необходимо иметь доступ к СУБД SQL Server. В этой книге широко используются инструментальные средства управления, которые входят в состав версии SQL Server 2005, поэтому автор настоятельно рекомендует применять дистрибутив, содержащий этот программный продукт в полном комплекте, а не просто дистрибутив SQL Server Express. Тем не менее в книге рассматриваются такие средства создания сценариев, которые в основном требуются для разработчиков, поэтому даже пользователи версии SQL Server Express смогут успешно освоить значительную часть материала, представленного в большинстве глав.

Для работы с этой книгой удобно пользоваться программой Visual Studio, но большинство необходимых для этого средств Visual Studio включено в программу Business Intelligence Studio, которая входит в поставку программного продукта SQL Server.

Удобные обозначения

В каждой главе книги используются удобные обозначения, позволяющие лучше усваивать излагаемый материал и проще находить наиболее важные сведения.

Практическое занятие Практическое задание

В разделах книги, обозначенных как практические задания, приведены упражнения, которые необходимо выполнить, следя за описанием в книге.

1. Практические задания обычно состоят из последовательности шагов.
2. Каждый шаг обозначается номером.
3. Для выполнения задания, как правило, используется существующая или создаваемая база данных.

Описание полученных результатов

Вслед за каждым разделом с практическим заданием приведено подробное описание того, какие результаты могут быть получены с помощью рассматриваемой процедуры.

Таким шрифтом выделена важная информация, о которой не следует забывать, непосредственно относящаяся к текущему изложению.

Подсказки, рекомендации, предложения и дополнительные сведения, которые имеют отношение к текущему обсуждению, обозначаются отступом и выделяются, как в этом примере.

Ниже приведены пояснения, касающиеся использования специальных шрифтов.

- Новые термины и словосочетания, имеющие особый смысл, при первом употреблении в тексте выделяются полужирным шрифтом.
- Отдельные клавиши и комбинации клавиш обозначаются так: <Ctrl+A>.
- Имена файлов, URL и элементы кода, представленные в тексте, выделены моноширинным шрифтом, например `persistence.properties`.
- Код, представленный отдельно от другого текста, выглядит так:

```
/* Комментарий, который
** занимает
** несколько строк
*/
SELECT * FROM Sales.Customer; -- Код и однострочный комментарий
```

Исходный код

Для практического выполнения примеров, приведенных в настоящей книге, можно либо набрать весь код вручную, либо воспользоваться файлами с исходным кодом, полученными с сопровождающего Web-узла книги. Весь исходный код, приведенный в книге, предоставляется для загрузки по адресу `http://www.wrox.com`. Перейдите на этот узел, введите в поле Search код ISBN англоязычного издания этой книги (`0-7645-8433-2`) или найдите в списке книг название *Beginning SQL Server 2005 Programming*, после чего щелкните на ссылке Download Code, которую можно найти на странице со сведениями об англоязычном издании настоящей книги, чтобы получить весь относящийся к ней исходный код.

Безусловно, поиск книги по названию является более трудоемким, поэтому рекомендуется использовать для поиска приведенный выше ISBN. Но следует учитывать, что после ввода в действие новой системы нумерации ISBN с 13 цифрами к январю 2007 года этот код изменится на `978-0-7645-8433-6`.

После загрузки архивного файла с исходным кодом остается только его разархивировать с помощью предпочтительного инструментального средства распаковки. Еще один вариант получения исходного кода состоит в том, что можно перейти на главную страницу загрузки кода на узле Wrox по адресу `http://www.wrox.com/dynamic/books/download.aspx`. На этой странице можно найти исходный код не только для настоящей книги, но и для всех других книг, выпущенных издательством Wrox.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

Адреса для писем:

из России: 115419, Москва, а/я 783

из Украины: 03150, Киев, а/я 152

1

Основные сведения о базах данных SQL Server

База данных предназначена для хранения данных, причем сама база данных функционирует под управлением СУБД (система управления реляционными базами данных). Но современные развитые реляционные СУБД не только обеспечивают хранение данных, но и позволяют управлять данными, регламентировать типы данных, которые могут быть введены в систему, а также упрощать процесс получения данных из системы. Если задача состоит лишь в том, чтобы сохранить данные в надежном месте, то достаточно воспользоваться практически любой системой хранения данных. Однако реляционные СУБД позволяют не только хранить данные, но и непосредственно задавать структуру данных, иными словами, устанавливать **бизнес-правила**, которым должны подчиняться данные.

Безусловно, бизнес-правила, в соответствии с которыми должны быть организованы данные, отличаются от более общих бизнес-правил, определяющих функционирование всей системы (например, согласно которым не следует предоставлять пользователю доступ к данным до тех пор, пока он не войдет в систему, или на основании которых отчетный период в системе бухгалтерского учета устанавливается на начало месяца). Дело в том, что правила, касающиеся функционирования самой системы, могут быть определены на любом уровне организации системы (но в наши дни обычно принято применять для управления работой всей системы средний, или клиентский, уровень многоуровневой системы). В данной книге речь в основном идет о бизнес-правилах, касающихся исключительно самих данных. В качестве примера можно привести правило, согласно которому в заказе на поставку не может быть задано отрицательное значение количества. Благодаря применению реляционной СУБД появляется возможность включать правила управления данными непосредственно в состав средств, обеспечивающих целостность самой базы данных.

В настоящей главе приведен общий обзор всей тематики, которая рассматривается более подробно в остальной части книги. Все вопросы, которые затрагиваются в этой главе, будут подробно описаны в следующих главах, но данная глава предназначена для использования в качестве руководства или плана, которого мы будем придерживаться, изучая последующий материал. Таким образом, в данной главе кратко описаны следующие темы:

- объекты базы данных;
- типы данных;
- другие средства базы данных, которые обеспечивают целостность данных.

Краткий обзор объектов базы данных

Реляционные СУБД, такие как SQL Server, состоят из многих **объектов**. Применительно к СУБД корпорация Microsoft использует термин *объект* в том смысле, который не соответствует определению объектов в объектно-ориентированных языках программирования. Ниже приведен список наиболее важных объектов SQL Server.

- Базы данных.
- Индексы.
- Журналы транзакций.
- Сборки.
- Таблицы.
- Отчеты.
- Файловые группы.
- Каталоги полнотекстового поиска.
- Диаграммы.
- Определяемые пользователем типы данных.
- Представления.
- Роли.
- Хранимые процедуры.
- Пользователи.
- Пользовательские функции.

Объект базы данных

Любая база данных по существу представляет собой объект наиболее высокого уровня, доступ к которому можно получить с помощью какой-либо конкретной СУБД SQL Server (с формальной точки зрения как объект может также рассматриваться сервер базы данных, но какие-либо практически применимые средства доступа к самому серверу отсутствуют, поэтому в дальнейшем он как объект не рассматривается). Бо́льшая часть всех прочих объектов SQL Server (но не все) являются дочерними по отношению к объекту базы данных.

Разработчики, знакомые с предыдущими версиями SQL Server, могут обнаружить, что в приведенном выше списке объектов отсутствуют учетные записи, удаленные серверы и задачи SQL Agent. Дело в том, что в СУБД SQL Server действительно имеется несколько других типов объектов (как указано выше), которые в основном предназначены для обеспечения функционирования самой базы данных. Но за исключением связанных серверов, а также, возможно, пакетов Integration Services, эти объекты в основном относятся к сфере деятельности администратора базы данных и поэтому, как правило, не становятся предметом значительного внимания в процессе разработки проектов приложений и самого программного обеспечения. Безусловно, для работы с подобными объектами могут применяться такие специальные средства, как интерфейс SMO (SQL Management Objects – объекты управления SQL), но это направление деятельности разработчиков является слишком специализированным для того, чтобы описывать его в книге для начинающих.

База данных, как правило, представляет собой группу объектов, которая включает по крайней мере набор объектов таблиц, а также чаще всего другие объекты, такие как хранимые процедуры и представления, относящиеся к определенной совокупности данных, которые хранятся в таблицах базы данных.

Таблицы, используемые в приложениях, могут находиться в одной базе данных или распределяться по нескольким базам данных. Причины, по которым может быть принято решение о распределении таблиц по разным базам данных, рассматриваются более подробно ниже в данной книге, но достаточно сказать, что в отдельной базе данных чаще всего имеет смысл хранить данные, относящиеся к какой-то одной прикладной системе или объединенные какими-то общими связями. Любая реляционная СУБД, в том числе SQL Server, позволяет обеспечить работу под управлением одного сервера нескольких пользовательских баз данных или поддерживать только одну базу данных. Количество пользовательских баз данных, подключенных к одному экземпляру SQL Server, зависит от таких факторов, как характеристики компьютерной системы (производительность процессора, ограничения по объему операций ввода-вывода на жестком диске. объем памяти и т.д.), требуемая степень автономности (связанная с предоставлением одному лицу прав на управление сервером, поддерживающим данную систему, а какому-то другому лицу — прав администратора применительно к другому серверу) или просто от того, сколько баз данных требуется для компании или клиента. На многих серверах эксплуатируется только одна база данных производственного назначения, а на других серверах таких баз данных может быть несколько. Кроме того, следует учитывать, что любая современная версия SQL Server, которую, скорее всего, можно найти в наши дни на производстве, предоставляет возможность эксплуатировать несколько экземпляров SQL Server (полностью с отдельными учетными записями и правами администрирования) на одном и том же физическом серверном компьютере. Дело в том, что версия SQL Server 2000 уже использовалась в течение пяти лет к тому времени, как началась ее замена, поэтому вполне можно предположить, что на большинстве предприятий в настоящее время используется эта или более современная версия.

Любопытно также отметить, что на одном и том же компьютере разрешается эксплуатировать одновременно несколько различных версий SQL Server, скажем, SQL Server 2000 и SQL Server 2005. Сам автор не рекомендует использовать подобную конфигурацию, даже на том этапе, когда осуществляется переход с одной версии на другую, но следует учитывать, что такая возможность существует.

Непосредственно после инсталляции СУБД SQL Server в состав программного обеспечения СУБД входят следующие системные базы данных:

- `master`;
- `model`;
- `msdb`;
- `tempdb`.

Для того чтобы сервер функционировал должным образом, необходимо инсталлировать все эти базы данных (в действительности эксплуатация сервера без некоторых из этих баз данных вообще невозможна). Кроме указанных баз данных, могут быть также установлены другие базы данных, в зависимости от того, в каких целях была выполнена инсталляция СУБД SQL Server. Чаще всего можно встретить примеры применения следующих баз данных:

- `AdventureWorks` (образцовая база данных);
- `AdventureWorksDW` (образец базы данных, применяемый в сочетании со службами Analysis Services).

Дополнительно к этим базам данных, которые могут быть установлены в процессе инсталляции системы, в данной книге рассматриваются также образцовые базы данных, относящиеся к более старым версиям, которые перечислены ниже. (Дополнительная информация об инсталляции этих баз данных приведена в приложении Д.)

- `pubs`.
- `Northwind`.

В ходе консультации с другими заинтересованными лицами по поводу проекта настоящей книги автору пришлось много спорить по поводу того, следует ли подготовить более свежие примеры или придерживаться проверенных и надежных примеров, подготовленных ранее. Безусловно, можно заранее предвидеть, что представителям корпорации Microsoft не слишком понравится мое решение сохранить более старые примеры, но я не собираюсь оправдываться по этому поводу.

Разумеется, появившаяся в последних версиях база данных `AdventureWorks` *представляет собой пример гораздо более совершенного объекта базы данных по сравнению с применявшимися ранее. Кроме того, база данных* `AdventureWorks` *позволяет проиллюстрировать почти любой нюанс и аспект использования версии SQL Server 2005. Но с этим связан единственный недостаток — значительная сложность. База данных* `AdventureWorks`*, рассматриваемая как учебная база данных, характеризуется чрезмерной усложненностью. В ней выбраны такие средства, которые, по всей видимости, должны использоваться лишь в исключительных случаях, но стали в ней ведущими. Я опросил нескольких своих друзей, которые занимаются преподаванием и (или) пишут книги по SQL Server, и все они согласились с моим мнением, что базы данных* `Northwind` *и* `pubs`*, хотя и во многом являются чрезмерно упрощенными, позволяют относительно легко понять основные принципы работы в СУБД SQL Server. А я намереваюсь, скорее, дать читателю понять основы и помочь достичь прогресса, чем ошеломить теми ненужными сложностями, которые присущи базе данных* `AdventureWorks`.

База данных master

База данных master входит в состав любой СУБД SQL Server, независимо от версии или специализированной модификации. Эта база данных содержит специальный набор таблиц (системные таблицы), которые позволяют отслеживать функционирование всей системы. Например, при создании на сервере новой базы данных в строку таблицы sysdatabases базы данных master вводится новая строка. Кроме того, в этой базе данных содержатся все расширенные и системные хранимые процедуры, независимо от того, для какой базы данных они предназначены. Итак, очевидно, что в базе данных master хранится почти вся информация, которая описывает конкретную инсталляцию, поэтому она является крайне необходимой для эксплуатации системы и не может быть удалена.

Кроме того, в трудную минуту могут оказаться чрезвычайно полезными системные таблицы, включая те, что находятся в базе данных master. В частности, системные таблицы позволяют определить, существуют ли конкретные объекты, прежде чем приступать к выполнению операций с этими объектами. Например, при попытке создать объект, который уже существует в какой-либо конкретной базе данных, возникает ошибка. Таким образом, если требуется создать объект, то можно проверить, существует ли этот объект, получив соответствующую запись из таблицы sysobjects, относящейся к рассматриваемой базе данных, и если объект действительно существует, удалить его, а затем создать снова.

В приведенной выше рекомендации по использованию одной из системных таблиц подразумевалось, что эта системная таблица применяется только для чтения. Так и следует всегда поступать. Использование системных таблиц в любой другой форме связано со значительной опасностью. Специалисты корпорации Microsoft предостерегают против непродуманного применения системных таблиц (такие предостережения можно найти в документации по меньшей мере трех последних версий SQL Server). К тому же нет абсолютно никаких гарантий в части совместимости баз данных master, относящихся к разным версиям, а в действительности даже гарантируется, что с выходом каждой версии в базе данных master происходят изменения. Тем более не следует вносить в системные таблицы какие-либо изменения, поскольку в результате этого полностью нарушается функционирование СУБД SQL Server. К счастью, для выборки основной части метаданных, хранящихся в системных таблицах, может применяться целый ряд других способов (например, основанных на использовании системных функций, системных хранимых процедур и представлений information_schema).

Несмотря на вышесказанное, в некоторых случаях практически невозможно поступить иначе. В настоящей книге будет описано несколько ситуаций, в которых невозможно избежать необходимости в непосредственном использовании системных таблиц, но, вообще говоря, их лучше всего оставить в покое.

База данных model

Назначение базы данных model полностью соответствует ее имени, поскольку она является моделью, с помощью которой создаются копии. База данных model используется как шаблон для любой вновь создаваемой базы данных. Это означает, что при желании в базу данных model можно внести изменения, если требуется, чтобы вновь создаваемые базы данных, соответствующие этому стандарту, выглядели иначе.

Например, в нее можно ввести набор таблиц аудита, которые в дальнейшем будут появляться в каждой создаваемой базе данных. Кроме того, можно также включить несколько групп пользователей, учетные записи которых появятся в каждой новой базе данных, создаваемой в системе. Итак, следует отметить, что база данных `model` служит в качестве шаблона для любой другой базы данных, поэтому является обязательной и должна оставаться в системе; эту базу данных удалять не разрешается.

При внесении изменений в базу данных `model` необходимо учитывать некоторые требования. Прежде всего, любая база данных, создаваемая на основе базы данных `model`, приобретает по меньшей мере такой же объем. Это означает, что после внесения изменений, согласно которым базы данных `model` приобретает объем 100 Мбайт, становится невозможным создание базы данных с объемом меньше 100 Мбайт. Необходимо учитывать также несколько аналогичных нюансов. Поэтому автор настоятельно рекомендует для 90% инсталляций оставлять эту базу данных неизменной.

База данных `msdb`

База данных `msdb` предназначена исключительно для хранения информации обо всех системных задачах, выполняемых программой **SQL Agent**. Если вы запланируете задачу создания резервных копий для какой-то базы данных, выполняемую каждую ночь, то в базе данных `msdb` появится строка. Такая строка вводится в базу данных `msdb` и после того, как будет запланировано одноразовое выполнение хранимой процедуры.

База данных `tempdb`

База данных `tempdb` представляет собой одну из основных рабочих областей для сервера. После вызова на выполнение любого сложного или крупного запроса, для осуществления которого в СУБД SQL Server потребуется создать промежуточные таблицы, такие таблицы создаются в базе данных `tempdb`. Временные таблицы появляются в базе данных `tempdb` и после того, как их создание запрашивает пользователь, даже если сам он полагает, что для этого используется текущая база данных. Каждый раз, когда возникает необходимость сохранить на время какие-либо данные, для решения такой задачи, по-видимому, используется база данных `tempdb`.

База данных `tempdb` во многом отличается от других баз данных. Временными являются не только создаваемые в ней объекты; сама база данных `tempdb` также существует лишь в течение определенного времени. Основное отличие базы данных `tempdb` от любых других баз данных, представленных в системе, состоит в том, что она полностью воссоздается с нуля после каждого перезапуска программного обеспечения SQL Server.

> С формальной точки зрения фактически разрешается также самим создавать объекты в базе данных `tempdb`, но автор настоятельно рекомендует не придерживаться такой практики. Достаточно предусмотреть создание временных объектов в любой базе данных системы, к которой вы имеете доступ, и в результате эти объекты будут сохранены в базе данных `tempdb`. Если же временные объекты создаются в базе данных `tempdb` самим пользователем, это не дает никаких преимуществ, но возникает путаница в отношении того, как следует обращаться к временным объектам из разных баз данных.

База данных AdventureWorks

Образцовые базы данных входили в состав программного обеспечения всех версий SQL Server, а не только последней. Тем не менее возможности таких образцов баз данных в предыдущих версиях были ограниченными. Еще одним их недостатком было то, что разработка этих баз данных не была выполнена в соответствии с лучшими рекомендациями по проектированию. (Автор не собирается втягиваться в спор по поводу того, преодолен ли этот недостаток в базе данных AdventureWorks. Достаточно сказать, что в базу данных AdventureWorks внесены значительные усовершенствования, кроме того, при ее разработке предприняты попытки устранить указанный недостаток.) К тому же образцы баз данных в предыдущих версиях были упрощенными и предназначались в основном для демонстрации отдельных концепций баз данных, а не иллюстрации возможностей SQL Server как программного продукта или самих баз данных в целом.

Специалисты корпорации Microsoft на самых ранних этапах разработки программного обеспечения Yukon (это – внутреннее кодовое название для программного продукта, который теперь известен как SQL Server 2005) знали, что им потребуется гораздо более надежный образец базы данных, который должен служить в качестве примера применения максимально возможного количества средств данного программного продукта. Результатом всех предпринятых для этого усилий стала база данных AdventureWorks. Сколько бы вам не пришлось выслушать от автора жалоб на чрезмерную сложность AdventureWorks для начинающего пользователя, эта база данных несомненно представляет собой настоящий шедевр, который демонстрирует все возможности современной версии СУБД SQL Server. Можно, безусловно, согласиться с тем, что эта база данных все же охватывает далеко не каждый возможный аспект применения SQL Server, но является довольно полным образцом, содержащим более реалистичные объемы данных по сравнению с другими образцами, включающим более сложные структуры данных и состоящим из разделов, которые демонстрируют примеры применения подавляющего большинства средств программного продукта. С этой точки зрения база данных AdventureWorks представляет собой буквально потрясающее достижение.

Сам автор постоянно использует примеры из базы данных AdventureWorks в повседневной работе. С дополнительной информацией по этой теме читатель ознакомится, когда приступит к изучению наиболее развитых средств вышеназванного программного продукта.

База данных AdventureWorksDW

База данных AdventureWorksDW представляет собой пример применения служб Analysis Services. В имени этой базы данных аббревиатура DW расшифровывается как Data Warehouse (хранилище данных); именно так характеризуется база данных того типа, на котором основано большинство проектов Analysis Services. По-видимому, самая замечательная особенность базы данных AdventureWorksDW состоит в том, что при ее создании специалисты корпорации Microsoft учли необходимость связать образец транзакционной базы данных с образцом аналитической базы данных и предоставили полный набор примеров, демонстрирующих совместное применение баз данных этих двух типов.

Описание баз данных, предназначенных для поддержки принятия решений, в основном выходит за рамки данной книги, поэтому примеры применения базы дан-

ных AdventureWorksDW в ней не рассматриваются, но о том, для чего предназначена эта база данных, всегда следует помнить, запустив на выполнение службу Analysis Services и проводя с ней эксперименты. Обратите внимание на различия между базами данных двух указанных типов. Они предназначены для обслуживания одной и той же фиктивной компании, но имеют разное назначение. Изучение данной темы будет весьма полезно.

База данных pubs

Тем, кто давно работает с программным обеспечением SQL Server, база данных pubs напоминает старого друга. Но в последнее время предусмотрена инсталляция pubs только в качестве отдельно загружаемого образца с Web-узла Microsoft, и в основном эта база данных предоставляется для использования в качестве средства иллюстрации статей и книг, предназначенных для применения во время учебы (таких как настоящая книга). База данных pubs абсолютно не требуется для обеспечения нормальной работы СУБД SQL Server. Она предназначена исключительно для использования в качестве постоянной отправной точки для обучения и экспериментирования. В настоящей книге также время от времени рассматриваются примеры, взятые из базы данных pubs.

Инсталляция pubs (хотя и должна проводиться отдельно) и ее удаление не влекут за собой каких-либо существенных последствий.

База данных Northwind

База данных Northwind хорошо знакома тем разработчикам, которые уже имеют опыт работы с программным обеспечением Access или Visual Basic. В составе программного обеспечения SQL Server база данных Northwind впервые появилась в версии 7.0, но была удалена из основной инсталляции со времени выхода версии SQL Server 2005. Эту базу данных, во многом аналогично pubs, необходимо устанавливать отдельно от базовой инсталляции SQL Server (но, к счастью, она входит в состав того же задания по загрузке и инсталляции образцов баз данных). База данных Northwind в настоящей книге служит в качестве одной из основных испытательных площадок.

> Базы данных pubs и Northwind могут быть установлены только с помощью отдельного инсталляционного пакета, который может быть загружен с Web-узла Microsoft. Дополнительная информация об инсталляции этих баз данных в прикладной системе приведена в приложении Д.

Журнал транзакций

Функционирование СУБД организовано так, что запись модифицированных данных не осуществляется непосредственно в файл базы данных. Безусловно, данные считываются из файлов базы данных, но любые изменения и дополнения в данных не передаются сразу же в саму базу данных. Вместо этого информация обо всех изменениях и дополнениях записывается в **журнал транзакций**. В какой-то последующий момент времени применительно к базе данных выполняется **контрольная точка**, и в этот момент времени все изменения и дополнения, зафиксированные в журнале, переносятся в физический файл (файлы) базы данных.

База данных представляет собой среду произвольного доступа, а журнал по своей организации является последовательным. Дело в том, что благодаря возможности применения к файлу базы данных операций произвольного доступа выборка данных ускоряется, а благодаря последовательному устройству журнала обеспечивается отслеживание операций обновления и вставки данных в должном порядке. Журнал накапливает информацию об изменениях и дополнениях, которые предназначены для дальнейшей фиксации, а затем время от времени осуществляется запись части накопленной информации в физический файл (файлы) базы данных.

Процесс ведения журнала рассматривается более подробно в главе 14, а пока достаточно запомнить, что первым делом данные попадают в журнал, который ведется на жестком диске, а в дальнейшем время от времени данные переносятся в действительную базу данных. Для того чтобы база данных могла функционировать, необходимо использовать и файл базы данных, и журнал транзакций.

Таблица как самый основной объект базы данных

Базы данных состоят из объектов многих разных типов, но ни один из этих объектов не является более важным для организации работы базы данных, чем таблица. Таблицу можно рассматривать как аналог бухгалтерского журнала или электронной таблицы Excel. Таблица состоит из так называемых **данных заголовка** (определений столбцов) и **данных тела** (самих строк). Все прикладные данные базы данных хранятся в таблицах.

Определение каждой таблицы содержит не только определения столбцов, но и **метаданные** (информацию, характеризующую данные), которые определяют характер данных, содержащихся в таблице. Определение каждого столбца представляет собой отдельный набор правил, касающийся того, что может храниться в этом столбце. Попытка нарушить правила, относящиеся к любому столбцу, может вынудить систему запретить выполнение операции вставки строки или обновления существующей строки или же предотвратить удаление строки.

Рассмотрим таблицу `publishers` базы данных `pubs`, фрагмент которой показан на рис. 1.1. (Изображение, представленное на рис. 1.1, — это копия части экрана, полученная в программе SQL Server Management Studio. Вышеназванная программа является одним из основных инструментальных средств, которое рассматривается в следующей главе.)

pub_id	pub_name	city	state	country
0736	New Moon Books	Boston	MA	USA
0877	Binnet & Hardley	Washington	DC	USA
1389	Algodata Infosy...	Berkeley	CA	USA
1622	Five Lakes Publis...	Chicago	IL	USA
1756	Ramona Publishers	Dallas	TX	USA
9901	GGG&G	M□nchen	*NULL*	Germany
9952	Scootney Books	New York	NY	USA
9999	Lucerne Publishing	Paris	*NULL*	France

Рис. 1.1. Фрагмент таблицы **`publishers`** *базы данных* **`pubs`**

Таблица, приведенная на рис. 1.1, состоит из пяти столбцов данных. Количество столбцов таблицы остается постоянным, независимо от того, сколько строк находится в таблице (даже если количество строк достигает нуля). В текущее время в таблице содержится восемь строк. Количество строк увеличивается и уменьшается по мере добавления и удаления данных, но характер данных в каждом поле (или столбце) каждой строки остается неизменным и соответствует определению **типа данных** столбца.

> В настоящей книге приведен целый ряд важных предостережений и это — первое из них. Оно касается именования объектов. В СУБД SQL Server предусмотрена возможность вставлять в имена пробелы, а в некоторых случаях — использовать в качестве имен ключевые слова. Но этого не следует делать. Безусловно, имена столбцов, содержащие пробелы, становятся удобными заголовками при выводе данных с помощью оператора `SELECT`, но такого же результата можно достичь другими способами. Применение имен столбцов, содержащих пробелы или представляющих собой ключевые слова, неизбежно приводит к возникновению программных ошибок, путаницы и других нарушений в работе. Информация о том, почему корпорацией Microsoft было принято решение допустить указанную возможность, будет приведена позже, но на данный момент достаточно помнить, что попытка использования в именах пробелов или ключевых слов влечет за собой большие неприятности (речь об этом пойдет также и в следующих главах).

Индексы

Индекс — это объект, который существует только в пределах инфраструктуры конкретной таблицы или представления. Индекс во многом аналогичен индексу (предметному указателю) научной книги. Он представляет собой набор поисковых (или ключевых) значений, отсортированных определенным образом. После создания индекса появляется возможность использования его в качестве ключа для поиска требуемой информации.

Индексы предназначены прежде всего для ускорения поиска информации. Индексы, применяемые в СУБД SQL Server, подразделяются на две описанные ниже категории.

- **Кластеризованный**. На каждой таблице может быть задан только один кластеризованный индекс. Если на таблице задан кластеризованный индекс, это означает, что строки таблицы с кластеризованным индексом отсортированы физически в соответствии с этим индексом. Например, если как индекс рассматривается предметный указатель научной книги, то подобный кластеризованный индекс состоит из номеров страниц, поскольку они показывают местонахождение различных сведений в научной книге на конкретных страницах.
- **Некластеризованный**. На каждой таблице может быть задано несколько некластеризованных индексов. Определение индекса этого типа еще больше соответствует тому, что представляет собой предметный указатель научной книги, поскольку индекс такого типа содержит информацию о дополнительных критериях, которые могут применяться для поиска данных. Применительно к рассматриваемому примеру с научной книгой в качестве индекса может рассматриваться предметный указатель, состоящий из ключевых слов.

Следует отметить, что представления, имеющие индексы (называемые также **индексированными представлениями**), должны иметь по меньшей мере один кластеризованный индекс, чтобы можно было определить на представлениях какие-либо некластеризованные индексы.

Триггеры

Триггер – это объект, который существует только в пределах инфраструктуры таблицы. Триггеры представляют собой фрагменты алгоритмического кода, автоматически вызываемые на выполнение в связи с тем, что к таблице, на которой они заданы, применяются определенные операции, такие как вставка, обновление или удаление.

Триггеры могут использоваться в самых различных формах, но главным образом применяются либо для копирования данных по мере их ввода, либо для проверки результатов обновления в целях определения того, соответствуют ли эти результаты заданным критериям.

Ограничения целостности

Ограничения целостности – это еще одна категория объектов, существующих только в связи с определенной таблицей. Ограничения целостности по своему назначению в основном соответствуют их названию, поскольку позволяют регламентировать ввод данных в таблицу в соответствии с определенными критериями. В определенном смысле ограничения целостности являются альтернативным средством обеспечения целостности данных по отношению к триггерам, но не следует считать их равнозначными. Ограничения целостности и триггеры как средства обеспечения целостности данных имеют свои преимущества и недостатки.

Файловые группы

По умолчанию все таблицы и прочие объекты базы данных (кроме журналов) хранятся в отдельных файлах. Каждый файл базы данных входит в состав так называемой **основной файловой группы**. Тем не менее в СУБД SQL Server обеспечивается применение других способов организации хранения данных.

В СУБД SQL Server предусмотрена возможность применять для хранения данных так называемые **вторичные файлы**, допустимое количество которых немного превышает 32 тысячи (по-видимому, этот предел определяется возможностями не самой СУБД SQL Server, а файловой системы). Такие вторичные файлы могут быть добавлены к первичной файловой группе или созданы в составе одной или нескольких **вторичных файловых групп**. Разумеется, предусмотрено применение только одной первичной файловой группы (которая так и называется – "`Primary`"), но можно также использовать до 255 вторичных файловых групп. Вторичные файловые группы создаются с помощью определенных опций оператора `CREATE DATABASE` или `ALTER DATABASE`.

Диаграммы

Вопросы формирования диаграмм баз данных будут рассматриваться более подробно в связи с описанием вопросов нормализации и проектирования базы данных, а на этот момент достаточно отметить, что диаграмма базы данных – это визуальное

представление проекта базы данных. Такое представление включает все таблицы, имена всех столбцов каждой таблицы и все связи между таблицами. Разработчикам программного обеспечения для баз данных часто приходится сталкиваться с диаграммами **сущность–связь**, или ER-диаграммами. В ER-диаграмме база данных представлена с помощью двух типов объектов: сущностей (таких как "поставщики" и "товары") и отношений (таких как "поставляет" и "приобретает").

В версии SQL Server 2005 предусмотрено много инструментальных средств проектирования базы данных, которые полностью изменились по сравнению с предыдущими версиями, но все равно количество таких инструментальных средств остается недостаточным. Кроме того, методология создания диаграмм, предусмотренная в этих инструментальных средствах, не соответствует ни одному из общепринятых стандартов формирования ER-диаграмм.

Тем не менее указанные инструментальные средства формирования диаграмм обеспечивают выполнение всех "обязательных" операций; по крайней мере, с их помощью можно приступить к освоению соответствующих методов. Дополнительная информация о программах создания ER-диаграмм и других инструментальных средствах приведена в приложении В.

На рис. 1.2 приведена диаграмма, на которой демонстрируется определенная часть из многочисленных таблиц базы данных `AdventureWorks`. Очевидно, что такая диаграмма не только показывает таблицы, но и описывает многие другие свойства базы данных (хотя для полного понимания смысла диаграмм требуется их более глубокое изучение). Обратите внимание на крошечные пиктограммы с изображениями ключей и знаков бесконечности. Эти пиктограммы описывают характер связи между таблицами. Более подробные сведения о таких связях приведены в главах 7 и 8, а также даны при описании других диаграмм в настоящей книге.

Представления

Представление — это своего рода виртуальная таблица. Кроме того, представления чаще всего используются полностью аналогично таблицам, хотя сами не содержат каких-либо данных. Представление фактически может рассматриваться как средство получения заранее запланированного отображения и преобразования данных, хранящихся в таблицах. Соответствующий план хранится в базе данных в форме запроса, а в запросе предусмотрена выборка данных из некоторых (но не обязательно из всех) столбцов, относящихся к одной или нескольким таблицам. При выборке данных могут применяться или не применяться (в зависимости от определения представления) критерии, которым должны соответствовать данные, содержащиеся в представлении.

В версиях, предшествующих SQL Server 2000, представления в основном предназначались для использования в качестве средства регламентации того, какие данные может просматривать пользователь. Применяемые при этом ограничения были направлены на достижение двух целей: обеспечение защиты и упрощение использования данных. Представления позволяют управлять процессом подготовки данных для просмотра пользователем. В частности, если к определенным столбцам таблицы (например, к сведениям о зарплате) должен быть предоставлен доступ только ограниченному кругу пользователей, то может быть создано представление, включающее лишь те столбцы, к которым разрешен доступ всем. Кроме того, представление может быть организовано таким образом, чтобы пользователям не приходилось заниматься поиском среди ненужной информации.

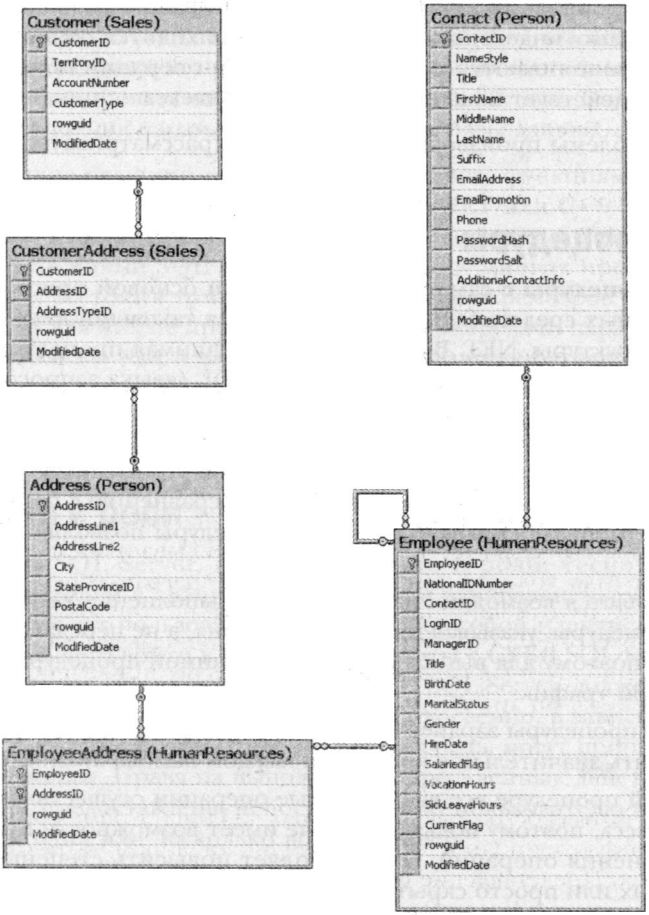


Рис. 1.2. Диаграмма, соответствующая части базы данных **`AdventureWorks`**

Выше были указаны наиболее важные области применения представлений, но предусмотрена также возможность создания так называемых **индексированных представлений**. Такие представления в основном не отличаются от всех прочих представлений, если не считать того, что они позволяют создавать индексы на представлениях. Как описано ниже, создание индекса оказывает определенное влияние на производительность (в основном положительное, а иногда и отрицательное).

- Представления, которые ссылаются на несколько таблиц, вообще говоря, обеспечивают более высокую производительность по сравнению с индексированными представлениями, поскольку соединения между таблицами формируются заранее.
- Результаты операций агрегирования, предусмотренных в определении представления, вычисляются заранее и сохраняются в составе индекса; это означает, что операция агрегирования выполняется единожды (при вставке или обновлении строки), а затем полученные результаты могут считываться непосредственно из информации индекса.

- Выполнение операций вставки и удаления связано с более высокими издержками, поскольку индекс на представлении приходится обновлять немедленно; издержки увеличиваются и при выполнении операции обновления, если последняя воздействует на ключевой столбец индекса.

Указанные проблемы производительности будут рассматриваться более подробно в главе 10.

Хранимые процедуры

Хранимые процедуры с самого начала служили основой программной реализации функциональных средств SQL Server и остаются таковыми даже в эпоху доминирования инфраструктуры .NET. Вообще говоря, хранимая процедура – это упорядоченная последовательность операторов Transact-SQL (языка запросов Microsoft SQL Server), оформленных в виде единого логического модуля. В хранимых процедурах допускается использование переменных и параметров, а также операторов управления ходом выполнения и циклических конструкций. Применение хранимых процедур предоставляет определенные преимущества по сравнению с передачей на сервер отдельных операторов, поскольку хранимые процедуры позволяют обеспечить следующее.

- Предоставляется возможность вызывать на выполнение непосредственно хранимые процедуры, указывая их короткие имена, а не передавать длинные строки текста, поэтому для выполнения кода хранимой процедуры требуется меньший сетевой трафик.
- Хранимые процедуры заранее оптимизируются и компилируются, что позволяет экономить значительное время при каждом вызове хранимой процедуры.
- В хранимой процедуре все выполняемые операции осуществляются в виде единого процесса, поэтому пользователь не имеет возможности проанализировать ход выполнения операций. Это позволяет повысить степень защищенности базы данных или просто скрыть от пользователя всю сложность ее устройства.
- Хранимые процедуры могут вызываться из других хранимых процедур, что позволяет обеспечить повторное использование кода, пусть даже в каком-то ограниченном смысле.

Кроме того, допускается возможность вводить в хранимые процедуры не только собственные конструкции языка T-SQL, но и программные конструкции любого языка .NET.

Пользовательские функции

Пользовательские функции (User Defined Functions – UDF) во многом аналогичны хранимым процедурам, но отличаются от них перечисленными ниже особенностями.

- Пользовательские функции могут возвращать значения, относящиеся к большинству типов данных SQL Server. Не допускается использовать в качестве типов возвращаемых значений лишь такие типы, как `text`, `ntext`, `image`, `cursor` и `timestamp`.

- Пользовательские функции не должны иметь побочных эффектов. По существу, в пользовательских функциях не допускается выполнение каких-либо действий, выходящих за пределы действия самой функции. Например, в них нельзя модифицировать таблицы, отправлять электронную почту и вносить изменения в значения параметров системы или базы данных.

Пользовательские функции во многом аналогичны функциям, используемым в обычном языке программирования, таком как VB.NET или C++. Эти функции принимают несколько параметров и возвращают одно значение. Различия между пользовательскими функциями SQL Server и функциями многих процедурных языков программирования состоят в том, что в них передача параметров осуществляется по значению, поэтому для них не предусмотрен способ передачи параметров, подобный применению ссылки `By Ref` в языке VB или передачи указателей в языке C++. Тем не менее пользовательские функции удобны тем, что позволяют возвращать данные в виде специальной таблицы. Дополнительная информация по этой теме приведена в главе 13.

Пользователи и роли

Такие объекты SQL Server, как пользователи и роли, тесно взаимосвязаны. Объекты **пользователей** в значительной степени эквивалентны учетным записям. Иными словами, объект пользователя представляет собой идентификатор для некоторого лица, желающего войти в систему для работы с СУБД SQL Server. Любой, кто пожелает зарегистрироваться для работы с СУБД SQL Server, должен быть представлен с помощью объекта пользователя (прямо или косвенно, в зависимости от применяемой модели защиты). Пользователи, в свою очередь, могут принадлежать к одной или нескольким **ролям**. Права на выполнение определенных действий в СУБД SQL Server могут быть предоставлены непосредственно пользователю или роли, к которой принадлежат один или несколько пользователей.

Правила

Правила и ограничения целостности предоставляют информацию, с помощью которой регламентируются возможности выполнения в таблице определенных действий. Если попытка выполнить операцию обновления или вставки строки приводит к нарушению какого-то правила, то соответствующая попытка вставки или обновления отвергается. Кроме того, правила могут использоваться для регламентации условий применения **типа данных, определяемого пользователем**. В отличие от правил, ограничения целостности в действительности не являются объектами, а, скорее, могут рассматриваться как фрагменты метаданных, описывающих конкретную таблицу.

Правила должны рассматриваться только как средство обеспечения обратной совместимости, поэтому их не следует применять при разработке приложений для новой версии SQL Server.

Значения, применяемые по умолчанию

Применяемые по умолчанию значения подразделяются на два типа. Во-первых, есть такие применяемые по умолчанию значения, которые представляют собой отдельные объекты, а во-вторых, предусмотрены и такие применяемые по умолчанию

значения, которые в действительности являются не объектами, а, скорее, метаданными, описывающими конкретный столбец в таблице (во многом аналогично тому, что существуют ограничения целостности, являющиеся метаданными, и правила, которые представляют собой объекты). Применяемые по умолчанию значения того и другого типа служат для одинаковой цели. Если при выполнении операции вставки строки не предоставляется значение для некоторого поля, а для столбца, соответствующего этому полю, задано применяемое по умолчанию значение, то автоматически происходит вставка этого применяемого по умолчанию значения. Применяемые по умолчанию значения обоих типов рассматриваются в главе 6.

Определяемые пользователем типы данных

Типы данных, определяемые пользователем, представляют собой дополнения к типам данных, определяемым системой. Начиная с рассматриваемой версии SQL Server возможности определяемых пользователем типов данных стали почти безграничными. Безусловно, применение определяемых пользователем типов данных было предусмотрено в SQL Server 2000 и в предыдущих версиях, но их действие в основном ограничивалось использованием различных способов ограничения по отношению к существующим типам данных. С другой стороны, в версии SQL Server 2005 предусмотрена возможность связывать сборки .NET с собственными типами данных, а это означает, что могут применяться такие типы данных, которые позволяют хранить почти любые данные, представимые в виде объектов .NET (безусловно, в пределах разумного).

При использовании указанных возможностей необходимо соблюдать осторожность. Дело в том, что тип данных, с которым вы работаете, является фактически основой представления и хранения данных. Безусловно, возможность определять собственные способы представления данных является очень привлекательной, но следует помнить, что для реализации этих возможностей почти наверняка потребуются большие затраты ресурсов. Поэтому тщательно продумывайте определения используемых типов данных, проверяйте, действительно ли они соответствуют заданным требованиям, а в дальнейшем тщательно контролируйте все, что связано с их использованием.

Каталоги полнотекстового поиска

Каталоги полнотекстового поиска представляют собой отображения данных, позволяющие ускорить поиск конкретных блоков текста в столбцах, для которых разрешен полнотекстовый поиск. Безусловно, эти объекты тесно связаны с таблицами и столбцами, для которых они служат отображениями, но представляют собой отдельные объекты и поэтому не обновляются автоматически при внесении изменений в базу данных.

Типы данных SQL Server

Выше в настоящей главе были описаны основные объекты базы данных SQL Server, а в этом разделе рассматриваются предусмотренные в SQL Server возможности определения одного из фундаментальных составляющих любой среды обработки данных — типов данных. Следует отметить, что эта книга предназначена для разра-

ботчиков, а ни один разработчик не сможет создать даже простейшее приложение, не имея знакомства с понятием типов данных, поэтому я предполагаю, что читатель уже знаком с тем, как применяются типы данных, и ему требуется лишь получить информацию о конкретных особенностях типов данных SQL Server.

В версии SQL Server 2005 предусмотрены встроенные типы данных, приведенные в табл. 1.1.

Таблица 1.1. Типы данных, предусмотренные в версии SQL Server 2005

Обозначение типа данных	Класс	Размер машинного представления в байтах	Описание и (или) пояснение
`bit`	Целочисленные данные	1	Сведения о размере машинного представления требуют пояснений. Один байт отводится для восьми элементов данных типа `bit` в таблице; если количество элементов данных такого типа меньше восьми, остальные биты байта не используются. Если же в столбце таблицы с типом данных `bit` допускается использование NULL-значений, то для представления этих значений применяются дополнительные байты
`bigint`	Целочисленные данные	8	Данные этого типа встречаются на практике все чаще и чаще, в связи с тем, что диапазон обрабатываемых значений постоянно возрастает. Данные типа `bigint` позволяют использовать целые числа от -2^{63} до $2^{63}-1$, что приблизительно соответствует положительному и отрицательному значениям в 92 квинтильона
`int`	Целочисленные данные	4	Целые числа от `-2 147 483 648` до `2 147 483 647`
`smallint`	Целочисленные данные	2	Целые числа от `-32 768` до `32 767`
`tinyint`	Целочисленные данные	1	Целые числа от `0` до `255`
`decimal` или `numeric`	Десятичные данные	Определяется отдельно	Заданная точность и масштаб от $-10^{38}-1$ до $10^{38}-1$. Обозначения `decimal` и `numeric` являются синонимами
`money`	Финансовые данные	8	Количество денежных единиц от -2^{63} до 2^{63}, определяемое с точностью до четырех десятичных позиций. Следует учитывать, что тип данных `money` позволяет представлять любые денежные единицы, а не только доллары
`smallmoney`	Финансовые данные	4	Денежные единицы от `-214 748.3648` до `+214 748.3647`
`float` (синоним для типа данных `real` по стандарту ANSI)	Приближенные числовые данные	Определяется отдельно	При определении данных этого типа допускается использовать параметр (например, `float(20)`), который определяет размер и, соответственно, точность. Следует учитывать, что параметр задается в битах. не байтах Область определения — от `-1.79E+308` до `1.79E+308`

Продолжение табл. 1.1

Обозначение типа данных	Класс	Размер машинного представления в байтах	Описание и (или) пояснение
`datetime`	Данные о дате и (или) времени	8	Данные о дате и (или) времени, которые относятся к периоду с 1 января 1753 года по 31 декабря 9999 года, определяемые с точностью до трех сотых секунды
`smalldatetime`	Данные о дате и (или) времени	4	Данные о дате и (или) времени, которые относятся к периоду с 1 января 1900 года по 6 июня 2079 года, определяемые с точностью до одной минуты
`cursor`	Специальные числовые данные	1	Указатель на курсор. Итак, для представления указателя на курсор требуется только один байт, но следует учитывать, что оперативная память необходима и для представления результирующего набора, который фактически образует курсор; точное значение количества необходимой оперативной памяти зависит от самого результирующего набора
`timestamp/ rowversion`	Специальные (двоичные) числовые данные	8	Специальное значение, которое является уникальным в пределах данной базы данных. Это значение устанавливается автоматически непосредственно в самой базе данных во время вставки или обновления каждой записи, даже если сам столбец с временной отметкой, типа `timestamp`, не упоминается в операторе `INSERT` или `UPDATE` (непосредственное обновление пользователем столбца с временной отметкой фактически не допускается)
`uniqueidentifier`	Специальные (двоичные) числовые данные	16	Специальный глобально уникальный идентификатор (Globally Unique Identifier — GUID). Уникальность любого идентификатора GUID в пространстве и времени является гарантированной
`char`	Символьные данные	Определяется отдельно	Символьные данные фиксированной длины. Значения данных с длиной короче заданной дополняются пробелами до указанной длины. Данные представлены в кодировке, отличной от Unicode. Максимальное заданное значение длины может составлять 8 000 символов
`varchar`	Символьные данные	Определяется отдельно	Символьные данные переменной длины. Значения данных с длиной короче заданной не дополняются пробелами. Данные представлены в кодировке, отличной от Unicode. Максимальное заданное значение длины может составлять 8 000 символов, но для обозначения длины можно использовать ключевое слово `max`, что фактически позволяет определять столбцы с символьными данными, имеющими чрезвычайно большой объем (до 2^{31} байтов данных)

Продолжение табл. 1.1

Обозначение типа данных	Класс	Размер машинного представления в байтах	Описание и (или) пояснение
`text`	Символьные данные	Определяется отдельно	Устаревший тип данных, который поддерживается в версии SQL Server 2005 исключительно для обеспечения совместимости с предыдущими версиями. Вместо этого типа данных следует использовать тип данных `varchar(max)`
`nchar`	Символьные данные в кодировке Unicode	Определяется отдельно	Символьные данные в кодировке Unicode фиксированной длины. Значения данных с длиной короче заданной дополняются пробелами. Максимальное заданное значение длины может составлять 4 000 символов
`nvarchar`	Символьные данные в кодировке Unicode	Определяется отдельно	Символьные данные в кодировке Unicode переменной длины. Значения данных с длиной короче заданной не дополняются пробелами. Максимальное заданное значение длины может составлять 4 000 символов, но для обозначения длины можно использовать ключевое слово `max`, что фактически позволяет определять столбцы с символьными данными, имеющими чрезвычайно большой объем (до 2^{31} байтов данных)
`ntext`	Символьные данные в кодировке Unicode	Определяется отдельно	Символьные данные в кодировке Unicode переменной длины. Как и тип данных `text`, этот тип данных является устаревшим и поддерживается в версии SQL Server 2005 исключительно для обеспечения совместимости с предыдущими версиями. В данном случае следует использовать тип данных `nvarchar(max)`
`binary`	Двоичные данные	Определяется отдельно	Двоичные данные фиксированной длины с максимальной длиной 8 000 байтов
`varbinary`	Двоичные данные	Определяется отдельно	Двоичные данные переменной длины с максимальной указанной длиной 8 000 байтов, но для обозначения длины можно использовать ключевое слово `max`, что фактически позволяет определять столбцы типа LOB, имеющие очень большой объем (до 2^{31} байтов данных)
`image`	Двоичные данные	Определяется отдельно	Этот тип данных является устаревшим и поддерживается в версии SQL Server 2005 исключительно для обеспечения совместимости с предыдущими версиями. В данном случае следует использовать тип данных `varbinary(max)`
`table`	Данные этого типа не подчиняются той же классификации, что и данные других классов	Определяется по особому принципу	Данные типа таблицы, `table`, предназначены прежде всего для использования в работе с результирующими наборами. Как правило, они передаются из пользовательских функций. Применение данных типа `table` в определениях таблиц не допускается

Окончание табл. 1.1

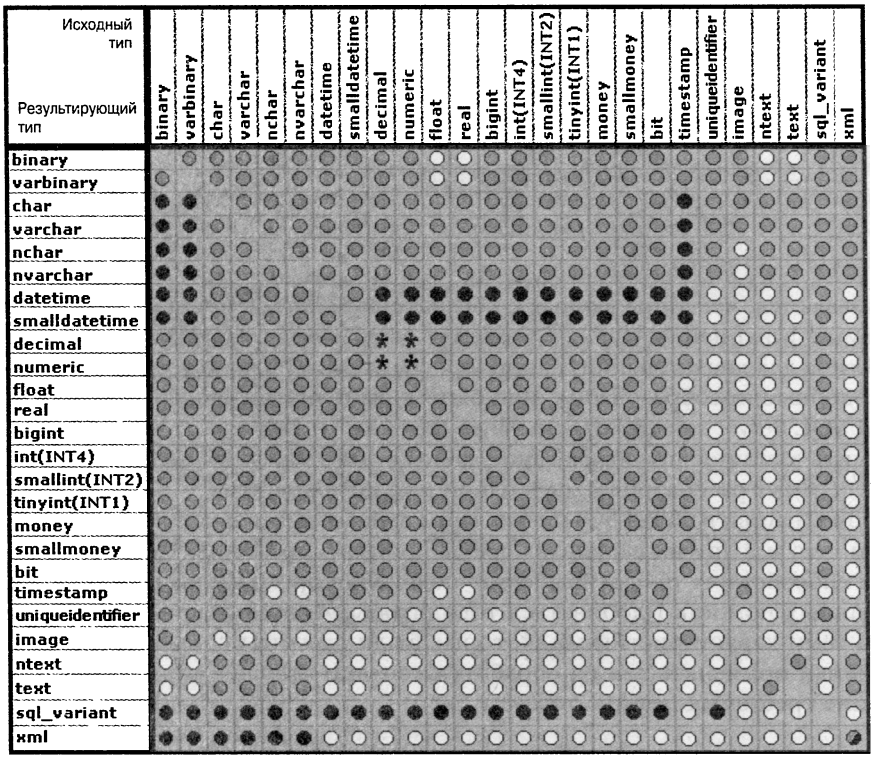
Обозначение типа данных	Класс	Размер машинного представления в байтах	Описание и (или) пояснение
`sql_variant`	Данные этого типа не подчиняются той же классификации, что и данные других классов	Определяется по особому принципу	Тип данных `sql_variant` может рассматриваться как приближенный аналог типа данных `Variant` языка VB и некоторых типов данных C++. По существу данные типа `sql_variant` представляют собой контейнер, который обеспечивает хранение большинства других типов данных SQL Server. Из этого следует, что тип данных `sql_variant` может использоваться, если необходимо представить в одном столбце или функции нескольких разных типов данных. Но, в отличие от типа данных `Variant` языка VB, при использовании типа данных `sql_variant` языка T-SQL требуется явно приводить эти данные к более определенному типу
`xml`	Символьные данные	Определяется отдельно	Определяет символьное поле как содержащее данные XML. Тип данных `xml` обеспечивает проверку данных по схеме XML и применение специальных функций, предназначенных для обработки кода XML

Большинство типов данных, представленных в табл. 1.1, имеют эквивалентные им типы данных в других языках программирования. Например, тип данных `int` в СУБД SQL Server является эквивалентным типу данных `Long` в языке Visual Basic, а также эквивалентен типу данных `int` в версиях языка C++, применяемых в большинстве комбинаций операционных систем и компиляторов.

> В СУБД SQL Server не предусмотрено применение числовых типов данных без знака.

Вообще говоря, типы данных SQL Server применяются в основном так же, как и в большинстве других современных языков программирования. Использование операции сложения к числам приводит к получению суммы, а после применения операции сложения к строкам происходит их конкатенация. Если в операции используются или присваиваются переменные или поля с разными типами данных, многие операции приведения данных к общему типу выполняются неявно (или автоматически), а для большинства других типов необходимо предусмотреть применение явно заданных операций преобразования (при этом должно быть конкретно указано, к какому типу данных должно быть приведено преобразуемое значение). Есть и такие типы данных, между которыми вообще невозможно выполнить преобразование. На рис. 1.3 приведена таблица, в которой показаны все возможные варианты преобразования данных из одного типа в другой.

Если читатель не уверен в том, что преобразование данных из одного типа в другой действительно требуется на практике, продемонстрируем такую необходимость на следующем простом примере.



● Явное преобразование

Неявное преобразование

○ Преобразование не допускается

* Требует применения функции CAST для предотвращения потери точности или масштаба, которая может возникнуть при неявном преобразовании

Неявные преобразования между типами данных XML поддерживаются, только если исходный тип или результирующий тип представляют собой нетипизированные данные xml. В противном случае преобразование должно быть явным

Рис. 1.3. Варианты преобразования данных T-SQL из одного типа в другой

Предположим, что необходимо вывести фразу "`Today's date is ##/##/####`", в которой вместо подстроки `##/##/####` должна быть приведена текущая дата. Безусловно, можно попытаться применить следующий оператор:

```
SELECT 'Today''s date is ' + GETDATE()
```

Операторы языка Transact-SQL, подобные приведенному выше, будут рассматриваться более подробно в ходе дальнейшего изложения материала настоящей книги, но читателю должно быть вполне очевидно, к каким результатам приводит выполнение приведенного оператора.

Проблема заключается в том, что попытка выполнить этот оператор приводит к получению следующего результата:

```
Msg 241, Level 16, State 1, Line 1
Syntax error converting datetime from character string.
```

Очевидно, что вместо строки с текущей датой получено сообщение об ошибке, поэтому попытаемся преобразовать дату с помощью функции CONVERT():

```
SELECT "Today's date is " + CONVERT(varchar(12), GETDATE(),101)
```

При этом должны быть получены примерно такие данные:

```
-----------------------------------
Today's date is 01/01/2000

(1 row(s) affected)
```

Оказывается, что типы данных с обозначением даты и времени, в частности, возвращаемые функцией GETDATE(), не могут быть явно преобразованы в данные строкового типа для конкатенации, например, со строкой "Today's date is", но на практике подобные преобразования приходится выполнять довольно часто. К счастью, в СУБД SQL Server предусмотрены функции CAST и CONVERT(), которые позволяют обеспечить взаимное преобразование данных многих разных типов. Функции CAST и CONVERT() будут рассматриваться в одной из следующих глав.

Короче говоря, в СУБД SQL Server типы данных применяются в основном для той же цели, что и в других вариантах среды программирования. Использование типов данных способствует предотвращению появления программных ошибок, поскольку типы данных помогают добиваться того, чтобы обрабатываемые данные обладали именно теми характеристиками, которые были приняты согласно исходному предположению, применять для их обработки соответствующие методы и обеспечивать получение требуемых результатов.

Неопределенные данные

Иногда на практике возникают такие обстоятельства, в которых при вводе строки отсутствуют данные, относящиеся к конкретному полю. Причиной этого может стать, в частности, то, что соответствующее значение просто неизвестно. В качестве примера можно указать такую ситуацию, что в базе данных компании имеется такая таблица, в отдельных строках которой ведется регистрация информации об итоговых результатах производственной деятельности за каждый год. А теперь предположим, что в один из столбцов, а именно, PercentGrowth, записываются данные (в процентах) о приросте за текущий год по сравнению с предыдущим. Но при вставке в базе данных самой первой строки данные об итогах за предыдущий год отсутствуют, поэтому невозможно установить, какое значение следует ввести в поле PercentGrowth. В частности, невозможно просто ввести в этом поле нулевое значение, поскольку это означало бы, что итоговый прирост за год был равен нулю, а это может не соответствовать действительности.

Для представления неопределенных данных используются специальное NULL-значение. Следует учитывать, что NULL-значение не имеет какого-то конкретного выражения, поскольку по определению подразумевается, что NULL-значение свидетельствует о том, что данные, вместо которых оно применяется, не определены, или, возможно, не применимы.

Идентификаторы объектов, применяемые в СУБД SQL Server

Выше в данной главе уже был приведен значительный объем информации об объектах, применяемых в СУБД SQL Server. Кроме того, автор уже высказал свое мнение о том, какими должны быть правильно выбранные имена столбцов. А в настоящем разделе приведены определенные сведения о способах именования объектов SQL Server.

Именуемые объекты SQL Server

По существу, в СУБД SQL Server имена приобретают все объекты. Частичный список именуемых объектов SQL Server приведен ниже.

- Хранимые процедуры.
- Таблицы.
- Столбцы.
- Представления.
- Правила.
- Ограничения целостности.
- Применяемые по умолчанию значения.
- Индексы.
- Файловые группы.
- Триггеры.
- Базы данных.
- Серверы.
- Пользовательские функции.
- Учетные записи.
- Роли.
- Каталоги полнотекстового поиска.
- Файлы.
- Определяемые пользователем типы.

Приведенный список является далеко не полным. Большинство объектов, которые только можно себе представить, кроме строк (которые в действительности не являются объектами), должны иметь имя. Задача разработчика заключается в том, чтобы имя, присвоенное им каждому объекту, было полезным и удобным.

Правила именования объектов

Как уже отмечалось выше, правила именования объектов SQL Server являются не слишком строгими; в именах допускается использовать пробелы и даже ключевые слова. Однако, как и большинство других свобод, свобода выбора имени оборачивается тем, что можно допустить злоупотребление и столкнуться с неприятностями.

Основные правила именования объектов приведены ниже.

- Имя объекта должно начинаться с любого символа, определенного в спецификации Unicode 2.0 как буква. К ним относятся, в частности, буквы `A-Z` и `a-z`, которые используются в письменных языках западных стран. Должны ли

прописные и строчные буквы рассматриваться как различные, зависит от настройки конфигурации сервера, но в качестве начальной буквы имени объекта допускается использовать и те и другие. Не считая обязательного требования об использовании буквы в первом символе имени объекта, больше нет практически никаких ограничений в части выбора остальных символов имени; допустимыми являются почти любые символы.

- Имена обычных объектов могут достигать длины 128 символов, а временных объектов — 116 символов.
- Любые имена, совпадающие с ключевыми словами СУБД SQL Server или содержащие пробелы, должны быть заключены в двойные кавычки ("") или квадратные скобки ([]). Состав слов, рассматриваемых как ключевые, зависит от уровня совместимости, установленного для базы данных.

Следует учитывать, что двойные кавычки допускается использовать в качестве разграничителя для имен столбцов, только если была задана соответствующая опция с помощью команды `SET QUOTED_IDENTIFIER ON`. *Таким образом, применение квадратных скобок позволяет исключить вероятность нарушения в работе, связанного с тем, что пользователи применяют неправильную настройку.*

Приведенные выше правила известны под общим названием *правил определения идентификаторов* и распространяются на любые объекты, для которых в программном обеспечении SQL Server используются имена. Для объектов определенных типов могут быть предусмотрены дополнительные правила.

> Следует еще раз подчеркнуть, что соблюдение рекомендации, согласно которой не следует использовать в именах объектов ключевые слова SQL Server или пробелы, является чрезвычайно важным. Разумеется, с формальной точки зрения допускается применение и тех и других, при условии соблюдения правил разграничения имен с помощью квадратных скобок, но, создавая и используя подобные имена, можно столкнуться с весьма значительными осложнениями.

Резюме

При проектировании реляционной базы данных, как и при решении большинства других практических задач, важно учитывать каждый нюанс. Чтобы создать действительно эффективную базу данных, недостаточно организовать хранение данных в таблицах. Следует также предусмотреть использование тех дополнительных возможностей, благодаря которым современные реляционные СУБД становятся столь мощными. Иными словами, необходимо ввести в действие такие объекты, которые позволяют реализовать непосредственно в базе данных необходимые функциональные средства и бизнес-правила.

При определении данных, хранящихся в базе данных, применяется тип, как и в большинстве других вариантов среды программирования. Тип данных должен в определенной степени учитываться при осуществлении большинства действий, связанных с эксплуатацией программного обеспечения SQL Server. Ознакомьтесь с имеющимися типами данных и постарайтесь сопоставить их с типами данных, применяемыми в знакомой вам среде программирования.

2

Доступные инструментальные средства

В предыдущей главе был приведен определенный объем информации о том, какие многочисленные типы объектов существуют в SQL Server, а в настоящей главе описано, как найти указанные объекты и как контролировать работу всей системы.

В данной главе рассматриваются инструментальные средства, которые входят в состав программного обеспечения SQL Server. Некоторые из этих инструментальных средств обеспечивают выполнение только небольшого числа весьма специализированных задач, а другие имеют более общее назначение. Бо́льшая часть рассматриваемых инструментальных средств входит в состав программного обеспечения SQL Server в течение продолжительного времени. Но в одном можно быть полностью уверенным — почти все программы, относящиеся к комплекту инструментальных средств SQL Server, при подготовке к выпуску версии SQL Server 2005 подверглись полной переработке. Подготавливая текущую версию, коллектив разработчиков инструментальных средств поставил перед собой основную цель проектирования — упростить поиск необходимых программ. Особенно полезными достигнутые при этом весомые результаты являются для начинающих разработчиков (разумеется, поиск необходимых инструментальных средств упростился и для опытных пользователей SQL Server).

В настоящей главе рассматриваются перечисленные ниже инструментальные средства.

- Документация SQL Server Books Online.
- Программа SQL Server Configuration Manager.
- Службы SSIS (SQL Server Integration Services), в том числе программа-мастер Import/Export Wizard.

- Программа bcp (Bulk Copy Program).
- Программа Profiler.
- Программа sqlcmd.

Будьте внимательны, обращаясь за помощью к разработчикам, которые имеют достаточный опыт, но используют SQL Server 2000 или более старую версию, а не SQL Server 2005. При подготовке последней версии комплект инструментальных средств подвергся весьма серьезной переработке, поэтому многие программы, которыми привыкли пользоваться "ветераны", давно работающие с СУБД SQL Server, перенесены в состав других инструментальных средств, включены в комплект программного обеспечения под другими именами или изъяты как самостоятельные программные продукты после их интеграции с другими инструментальными средствами.

Большинство существующих служебных программ все еще можно найти в составе того или другого инструментального средства, но, возможно, уже не на прежнем месте и под другим именем.

Документация Books Online

Следует ли рассматривать документацию **Books Online** как инструмент разработчика? По мнению автора, ответ на этот вопрос должен быть положительным. Дело в том, что, прочитав даже несколько раз эту и любую другую книгу по SQL Server, вы не сможете запомнить все, что когда-либо может потребоваться в работе с программным обеспечением SQL Server. Сам автор повседневно работает с СУБД SQL Server и все равно не может утверждать, что сумел сохранить в памяти все сведения об этой системе. Поэтому документация Books Online — это не что иное, как наиболее важное инструментальное средство, которое можно найти в составе программного обеспечения SQL Server.

Сам я рассматриваю книги и любые другие справочные материалы, относящиеся к программированию, под таким углом зрения, что просто невозможно сосредоточить в одних руках достаточное количество подобных источников информации. Впервые я начал заниматься разработкой программного обеспечения примерно в 1980 году. В то время было возможно удерживать в памяти основную часть необходимых сведений (но не все), а сейчас это просто невозможно. Если же вы заняты в нескольких областях деятельности (что само по себе в настоящее время является довольно сложным), то приходится запоминать слишком многое, и те сведения, к которым не приходится возвращаться повседневно, постепенно забываются.

Примите простой совет — не старайтесь все запомнить. Старайтесь удерживать в памяти то, что является неотъемлемой частью вашей деятельности и с чем вам приходится работать повседневно. И не упускайте из виду то, что необходимо постепенно создавать хорошую справочную библиотеку (начиная с этой книги), чтобы можно было извлечь из нее другие нужные данные.

В документации Books Online из состава программного обеспечения SQL Server используется обновленный оперативный справочный интерфейс .NET, который заменил применявшийся ранее стандартный оперативный справочный интерфейс, предназначенный для специализированных программных продуктов Microsoft (Back Office, MSDN и Visual Studio). Внешний вид интерфейса Books Online показан на рис. 2.1.

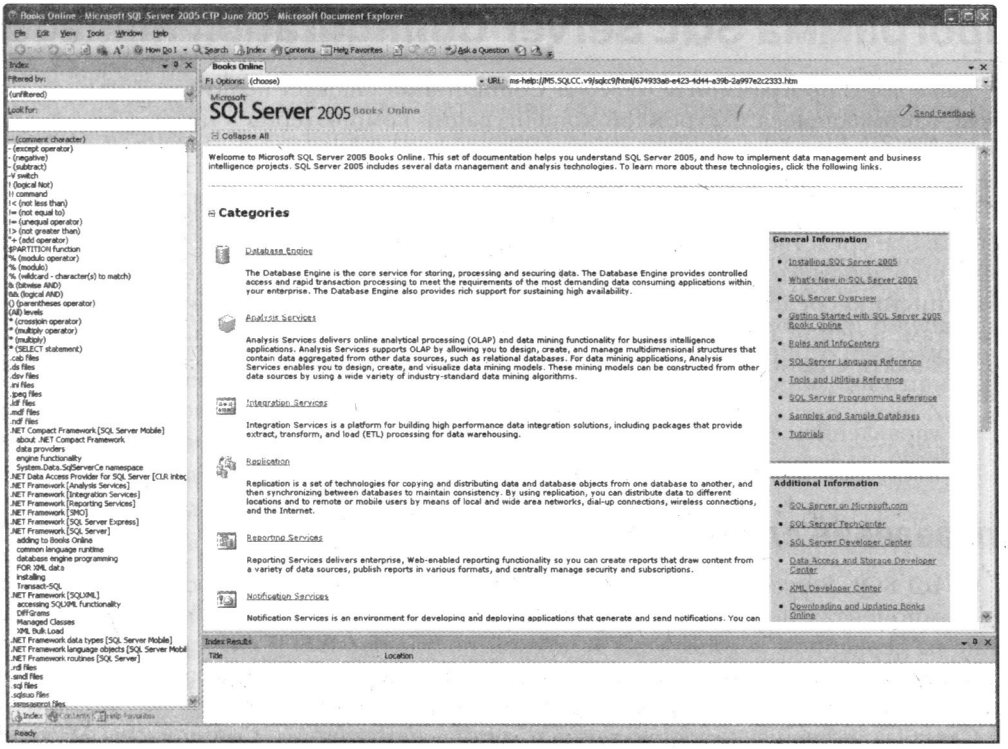


Рис. 2.1. Внешний вид интерфейса Books Online

Все компоненты справочного интерфейса Books Online действуют в основном в соответствии с ожиданиями, поэтому в данной главе не приводятся подробные сведения о том, как эксплуатировать справочную систему. Достаточно отметить, что документация SQL Server Books Online представляет собой превосходный быстродействующий справочник, к которому всегда можно обратиться с любого компьютера, за которым вы работаете. Еще одним преимуществом документации Books Online является то, что в ней чаще всего можно найти более свежую информацию по сравнению с той, что приведена даже в вышедшей недавно печатной документации.

Вполне возможно, что документация Books Online (BOL) будет установлена не в каждой системе, с которой вам доведется работать. Причина этого заключается в том, что во время инсталляции вручную отменяется опция установки BOL. Но автор настоятельно рекомендует всегда устанавливать BOL, даже в условиях недостаточного объема свободного пространства. В действительности инсталляция этой документации обходится не так уж дорого, с учетом того, что в наши дни не требуются большие затраты на приобретение дискового пространства, а если в вашем распоряжении в процессе эксплуатации СУБД SQL Server есть быстродействующий справочник, то вы всегда сможете найти выход из любого положения (на компьютере автора документация Books Online занимает около 100 Мбайт памяти на диске).

Программа SQL Server Configuration Manager

Чаще всего программой SQL Server Configuration Manager пользуются администраторы, занимающиеся настройкой конфигурации компьютеров в целях обеспечения доступа к базе данных, но это инструментальное средство позволяет получить также некоторые дополнительные возможности.

Программа SQL Server Configuration Manager представляет собой новое инструментальное средство, которое вошло в состав программного обеспечения SQL Server 2005, но в действительности создание этой программы стало результатом усилий по объединению средств настройки, которые были разбросаны по многочисленным утилитам, в одном приложении. Опции настройки конфигурации, которыми можно управлять с помощью программы Configuration Manager, подразделяются на два типа:

- управление службами;
- настройка конфигурации сети.

Управление службами

СУБД SQL Server – это крупный программный продукт, поэтому в различных его компонентах используется широкий набор служб, работающих в фоновом режиме на серверном компьютере. В полной инсталляции SQL Server предусматривается установка семи служб, и соответствующий компонент программы SQL Server Configuration Manager позволяет управлять всеми этими службами.

Службы, доступные для управления с помощью программы Configuration Manager, перечислены ниже.

- Analysis Services. Эта служба, в соответствии со своим названием, обеспечивает функционирование машины Analysis Services.
- Full Text. Эта служба также соответствует своему названию; она обеспечивает работу машины полнотекстового поиска, Full Text Search Engine.
- Report Server. Основная машина, которая поддерживает службы Report Services.
- SQL Server Agent. Основная машина, обеспечивающая планирование выполнения всех задач SQL Server. С помощью данной службы могут быть запланированы на выполнение задания путем включения их в различные расписания. В свою очередь, задания могут состоять из многочисленных задач и даже предусматривать переход от одной задачи к другой по условию, в зависимости от результатов выполнения предыдущей задачи. К примерам операций, осуществляемых с помощью службы SQL Server Agent, относится резервное копирование, а также выполнение повседневных задач импорта и экспорта.
- SQL Server. Основная машина базы данных, которая обеспечивает доступ к средствам хранения данных, выполнение запросов и настройку конфигурации системы SQL Server.
- SQL Server Browser. Служба, которая обеспечивает анонсирование информации о сервере, для того чтобы пользователи, просматривающие ресурсы локальной сети, могли определить, на каком компьютере установлено программное обеспечение SQL Server.

Настройка конфигурации сети

Проблемы нарушения связи, возникающие в сети, чаще всего являются результатом неправильной настройки конфигурации сети на клиентском компьютере или вызваны несоответствием параметров конфигурации, заданных на клиентском и серверном компьютерах.

В состав программного обеспечения SQL Server входит целый ряд так называемых **сетевых библиотек**, которые сокращенно обозначаются **NetLib** (Net-Library). Библиотеки NetLib представляют собой динамически связываемые библиотеки (Dynamic-Link Library – DLL), которые используются в СУБД SQL Server для обеспечения обмена данными с помощью определенных **сетевых протоколов**. Библиотеки NetLib могут рассматриваться в качестве своего рода промежуточного программного обеспечения, обеспечивающего взаимодействие клиентских и серверных приложений по сети. Сетевые протоколы, которые поддерживаются библиотеками NetLib, предусмотренными в версии SQL Server 2005, перечислены ниже.

- Протоколы Named Pipes.
- Протоколы TCP/IP (применяются по умолчанию).
- Протоколы сетевого взаимодействия с помощью разделяемой памяти (Shared Memory).
- Протоколы VIA.

VIA – это специальная сетевая библиотека, предназначенная для использования с некоторым весьма специализированным (и весьма дорогостоящим) аппаратным обеспечением. Для обеспечения работы в среде VIA необходимо выполнить специальные требования, которые к ней относятся. А тем, кто не работает в этой среде, достаточно знать, что VIA представляет собой комплект чрезвычайно быстродействующих, но дорогостоящих программных и аппаратных средств, предназначенных для обеспечения высокоскоростной связи между серверами. Как правило, библиотека VIA не используется для подключения обычных клиентских компьютеров.

Для того чтобы клиентский и серверный компьютеры могли обмениваться данными друг с другом с помощью сетевого протокола, на каждом из них должны быть установлены одинаковые библиотеки NetLib. Если на клиентском компьютере выбрана библиотека NetLib, не поддерживаемая на серверном компьютере, то попытка установления соединения оканчивается неудачей и возникает ошибка `Specified SQL Server Not Found`.

В состав сетевого программного обеспечения обязательно входит драйвер, который взаимодействует с библиотекой NetLib. Выбор конкретного драйвера зависит от применяемого метода доступа к данным и типа драйвера (SQL Native Client, ODBC, OLE DB или DB-Lib). Схема процесса сетевого взаимодействия показана на рис. 2.2. Этапы процесса организации сетевого взаимодействия перечислены ниже.

1. Клиентское приложение вызывает сетевой драйвер (SQL Native Client, ODBC, OLE DB или DB-Lib).
2. Драйвер вызывает клиентскую библиотеку NetLib.
3. Библиотека NetLib обращается к драйверу соответствующего сетевого протокола и передает данные в серверную библиотеку NetLib.
4. Серверная библиотека NetLib передает запросы от клиента к СУБД SQL Server.

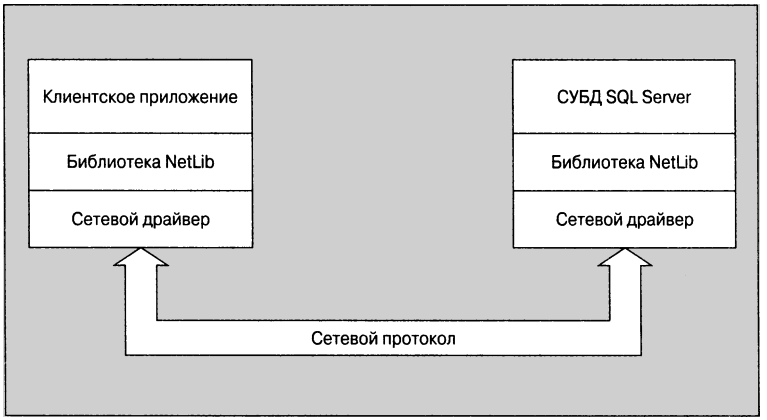


Рис. 2.2. Схема процесса сетевого взаимодействия

Ответы, передаваемые от сервера SQL Server к клиенту, следуют тем же путем, но осуществляемые при этом действия выполняются в обратном порядке.

> По умолчанию в библиотеке NetLib протокола TCP/IP для приема запросов применяется сетевой порт 1433.

Протоколы

Вначале рассмотрим перечень доступных протоколов. После вызова на выполнение программы Computer Management Utility и развертывания дерева Server Network Configuration открывается окно, показанное на рис. 2.3.

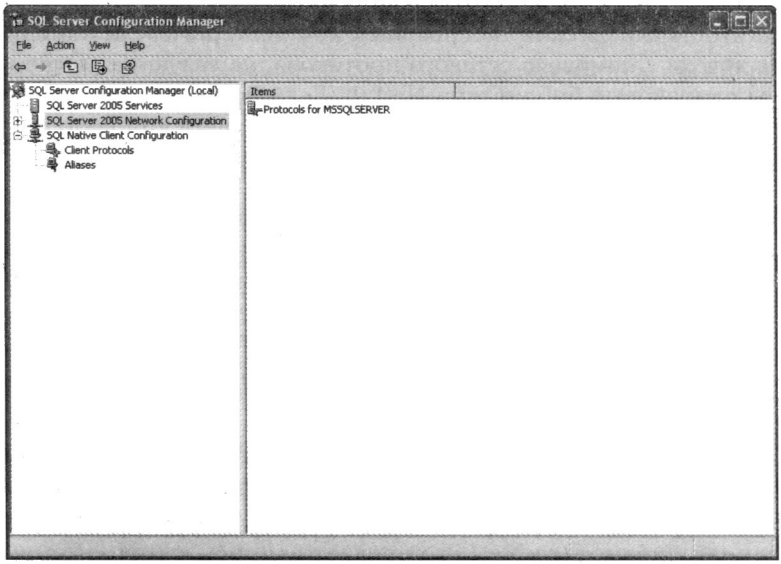


Рис. 2.3. Окно SQL Server Configuration Manager

По умолчанию разрешено применение только протокола доступа к разделяемой памяти, Shared Memory. В предыдущих версиях программного продукта SQL Server по умолчанию было разрешено использование тех или других библиотек NetLib в зависимости от версии SQL Server и операционной системы.

Протокол доступа к разделяемой памяти позволяет обеспечить доступ к СУБД SQL Server, установленной на том же компьютере, где находится клиентское программное обеспечение. Если же требуется обеспечить взаимодействие, например, Web-сервера или различных клиентских программ в сети с СУБД SQL Server, установленной на другом компьютере, то необходимо инсталлировать по крайней мере еще одну библиотеку NetLib.

Чтобы узнать, по каким протоколам сервер может принимать запросы на установление соединения, необходимо развернуть узел Protocols for MSSQLSERVER, находящийся под узлом SQL Server 2005 Network Configuration (рис. 2.4).

Следует учитывать, что клиент может установить соединение с сервером, только если сервер принимает запросы на установление соединения по такому же протоколу, по какому пытается с ним взаимодействовать клиент. При этом в клиентском программном обеспечении должен быть правильно указан порт. Поэтому если в сети используется программное обеспечение Named Pipes, то необходимо ввести в действие библиотеку Named Pipes. Для этого следует снова перейти к дереву Protocols, щелкнуть правой кнопкой мыши на обозначении протокола Named Pipes и выбрать команду Enable.

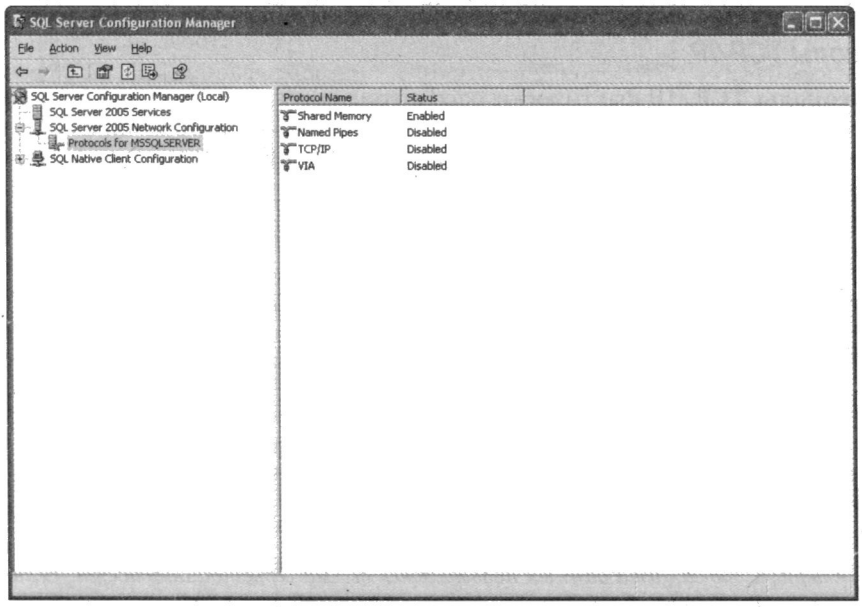


Рис. 2.4. Узел Protocols for MSSQLSERVER

Напрашивается вопрос, почему бы сразу не разрешить применение всех возможных библиотек NetLib? В таком случае не пришлось бы задумываться над тем, введена ли в действие нужная библиотека. Но при этом необходимо учитывать ту же рекомендацию, которая касается дополнительного подключения к серверу любых компонентов, — не следует устанавливать ненужные компоненты, поскольку из-за них увеличиваются издержки. В данном случае ввод в действие неиспользуемых протоколов приведет к замедлению работы сервера (не очень значительному, но здесь важна каждая мелочь). К тому же появятся дополнительные возможности проникновения в систему, что может привести к нарушению защиты (зачем оставлять еще одну дверь открытой, если ею никто не собирается пользоваться).

Ниже рассматриваются поддерживаемые протоколы и приведены причины, по которым следует выбирать тот или иной протокол.

Протокол Named Pipes

Протокол Named Pipes может оказаться очень полезным, если не применяются протоколы TCP/IP или отсутствует сервер DNS (Domain Name Service – служба доменных имен), позволяющие находить серверы в сети TCP/IP по именам.

> Говоря формально, имеется еще возможность подключиться к серверу SQL Server с помощью протоколов TCP/IP, указав IP-адрес этого сервера вместо имени. Такой способ организации доступа всегда остается применимым, даже если не функционирует служба DNS и не применяются другие средства преобразования сетевых имен в сетевые адреса, при условии, что существует маршрут от клиента к серверу (если задан IP-адрес, то имя не требуется, но имена применяются ради удобства).

Протоколы TCP/IP

Протоколы TCP/IP фактически рассматриваются как стандартный набор сетевых протоколов и применяются по умолчанию в составе программного обеспечения SQL Server, начиная с версии SQL Server 2000. Кроме того, если требуется обеспечить подключение клиентской программы к СУБД SQL Server через сеть Интернет (в которой, безусловно, используется только протокол IP), то единственный вариант состоит в использовании набора протоколов TCP/IP.

Не следует путать задачу обеспечения взаимодействия сервера базы данных с Web-сервером и задачу предоставления доступа к серверу базы данных через Интернет. В определенных конфигурациях программного обеспечения может быть предусмотрено применение Web-сервера, доступ к которому предоставляется через Интернет, притом что Web-сервер взаимодействует с сервером базы данных, непосредственный доступ к которому через Интернет невозможен (в таком случае единственный способ получения доступа к данным с помощью сервера базы данных состоит в использовании соединения Интернет, с помощью которого осуществляется взаимодействие с Web-сервером).

Если предусмотрена возможность подключения к серверу базы данных непосредственно через Интернет, то защита данных подвергается весьма серьезной опасности. Если все-таки приходится применять такую организацию работы (и на это есть серьезные причины), то следует уделять особое внимание применению всех предосторожностей против нарушений защиты.

Протокол Shared Memory

Если используется протокол доступа к разделяемой памяти (Shared Memory), то исключается необходимость преобразовывать информацию, которой обмениваются клиентская и серверная программы, в форму, применимую для передачи по сети, поскольку обе эти программы эксплуатируются на одном и том же компьютере. Клиентская программа имеет непосредственный доступ к тому же файлу, отображаемому в память, в который серверная программа выводит данные. В результате этого исключаются значительные издержки и быстродействие существенно повышается. Безусловно, такая организация работы применима, только если клиентское приложение обращается к серверу локально (скажем, с помощью Web-сервера, инсталлированного на том же серверном компьютере, где находится база данных), но при этом могут достигаться существенные преимущества с точки зрения производительности.

Применение протоколов в клиентском приложении

Выше в данной главе описаны все применимые протоколы и показано, по каким критериям может осуществляться выбор среди этих протоколов. Определив перечень протоколов, поддерживаемых сервером, можно переходить к настройке конфигурации клиента. Чаще всего вполне удовлетворительными являются значения параметров настройки, заданные по умолчанию, поэтому рассмотрим, какие результаты достигаются при их использовании. Разверните поддерево SQL Native Client Configuration и выберите узел Client Protocols, как показано на рис. 2.5.

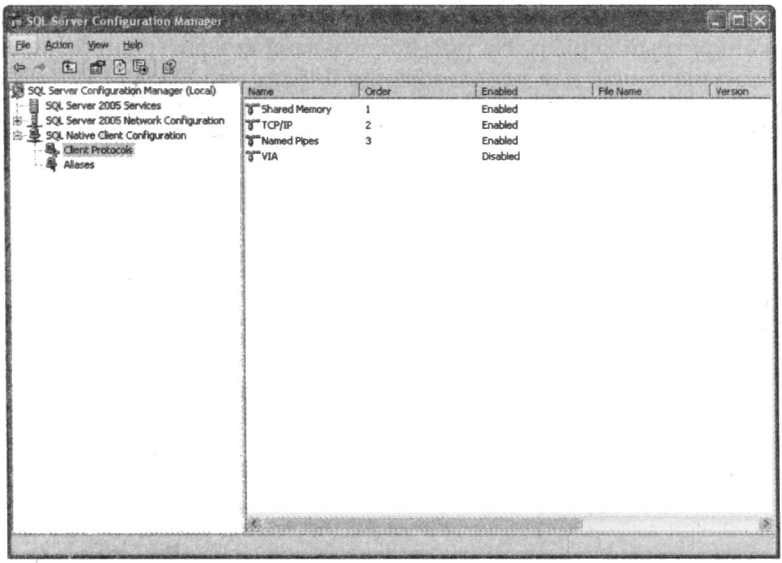


Рис. 2.5. Узел Client Protocols

Начиная с версии SQL Server 2000 корпорацией Microsoft была предусмотрена возможность осуществлять запуск клиентской программы с одним протоколом, а затем, если он окажется неработоспособным, переходить к другому протоколу. В диалоговом окне, приведенном на рис. 2.5, видно, что вначале предусмотрено использование протокола Shared Memory, после чего предпринимается попытка применить TCP/IP

и наконец осуществляется переход к протоколу Named Pipes, если TCP/IP не работает. Такая последовательность определена в столбце Order. Этот порядок выбора протоколов задан по умолчанию, и если он не будет изменен (путем смены приоритетов по принципу перемещения строк с обозначениями протоколов вверх или вниз с помощью клавиш со стрелками), то в первую очередь для создания соединений с любым сервером, не перечисленным в списке псевдонимов Aliases (узел, следующий за Client Protocols), используется библиотека NetLib, относящаяся к протоколу Shared Memory, затем применяется библиотека для TCP/IP и т.д.

> Если в сети предусмотрена поддержка набора протоколов TCP/IP, то следует оставить параметры настройки конфигурации сервера неизменными, что влечет за собой применение именно этого набора протоколов. Протоколы TCP/IP налагают меньше издержек и обладают более высоким быстродействием, поэтому единственной причиной отказа от их использования может быть отсутствие поддержки этих протоколов в сети. Но следует отметить, что для локальных серверов (таковым называется сервер, эксплуатируемый на том же физическом компьютере, что и клиент) библиотека NetLib протокола Shared Memory обеспечивает более высокое быстродействие, поскольку для обеспечения обмена данными между клиентом и локальным сервером SQL не нужно выходить в сеть.

Список Aliases содержит перечень всех серверов, для которых определена конкретная библиотека NetLib, применяемая при обмене данными с тем или иным конкретным сервером. Это означает, что для клиентской программы может быть предусмотрена возможность взаимодействовать с одним протоколом с помощью TCP/IP, а с другим — с помощью Named Pipes; выбор протокола определяется тем, что*о* поддерживает конкретный сервер. На рис. 2.6 показана конфигурация клиентской программы, предназначенная для использования библиотеки NetLib типа Named Pipes для обмена данными с сервером `Aristotle`; при этом для взаимодействия со всеми другими СУБД SQL Server применяются протоколы, которые заданы по умолчанию.

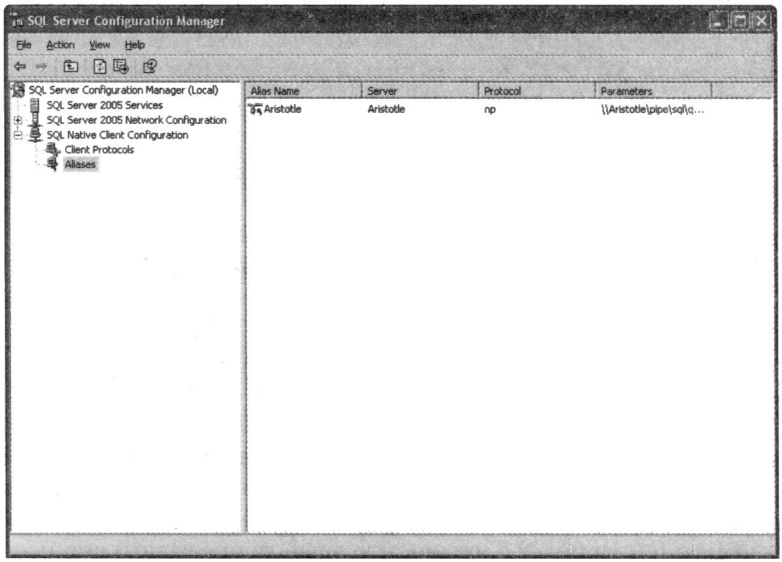


Рис. 2.6. Конфигурация клиентской программы

Еще раз отметим, что в составе параметров настройки конфигурации SQL Native Client Configuration может быть задан хотя бы один применяемый по умолчанию протокол, который соответствует одному из протоколов, поддерживаемых сервером, поскольку в противном случае необходимо ввести запись в список Aliases и конкретно указать библиотеку NetLib, поддерживаемую сервером.

Если предусмотрено подключение к СУБД SQL Server по Интернету (что весьма не рекомендуется по соображениям защиты, но многие именно так и поступают), то, по-видимому, целесообразно использовать действительный IP-адрес сервера, а не задавать имя сервера. Это позволяет исключить некоторые проблемы преобразования имен, которые могут возникнуть при обеспечении доступа к СУБД SQL Server через Интернет. Но следует учитывать, что после получения сервером нового IP-адреса потребуется изменить данный IP-адрес вручную, поскольку нельзя будет рассчитывать на то, что система DNS автоматически преобразует имя сервера в IP-адрес.

Программа SQL Server Management Studio

Программа **SQL Server Management Studio** является одним из основных программных средств администрирования СУБД SQL Server. В этой программе предусмотрен целый ряд функциональных средств управления сервером, в которых применяется относительно простой в использовании графический интерфейс пользователя.

Программа Management Studio была впервые введена в версии SQL Server 2005. Эта программа, интерфейс которой немного напоминает интерфейс среды интегрированной разработки DevStudio, объединяет в себе бесчисленное множество функциональных возможностей, которые прежде были реализованы в составе отдельных инструментальных средств.

Рассмотрение всех операций, которые могут быть осуществлены с помощью программы Management Studio, выходит за рамки данной книги, и ниже приведен лишь краткий обзор действий, выполняемых с помощью этой программы.

- Создание, модификация и удаление базы данных и объектов базы данных.
- Управление планируемыми задачами, такими как резервное копирование, и обеспечение прогона пакетов SSIS.
- Отображение данных о текущем состоянии функционирования базы данных, в частности, о том, какие пользователи в ней зарегистрированы, какие объекты заблокированы и из какой клиентской программы запущены те или другие процессы.
- Управление средствами защиты, включая определение таких составляющих защиты, как роли, учетные записи, удаленный и связанные серверы.
- Инициализация и управление почтовой службой базы данных, Database Mail Service.
- Создание и управление каталогами полнотекстового поиска.
- Управление параметрами настройки конфигурации для сервера.
- Создание и управление базами данных публикации и подписки, применяемыми для репликации.

В следующих главах данной книги приведен большой объем информации о программе Management Studio, а в настоящей главе рассматриваются некоторые наиболее важные функции, осуществляемые с ее помощью.

Вызов программы Management Studio на выполнение

При первом запуске программы Management Studio отображается диалоговое окно Connect to Server (рис. 2.7).

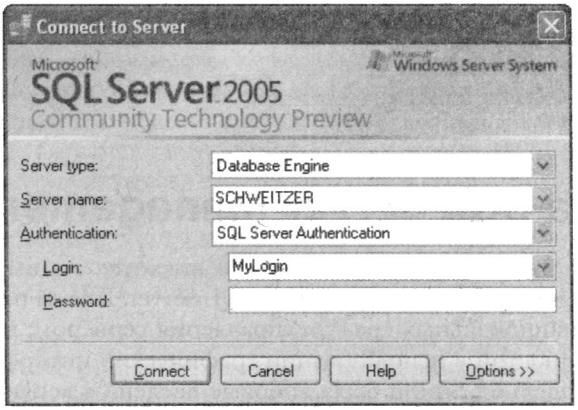


Рис. 2.7. Диалоговое окно Connect to Server

Внешний вид окна входа в систему (рис. 2.7) может оказаться другим, в зависимости от того, был ли до этого выполнен вход в систему, на каком компьютере осуществляется регистрация пользователя в системе и какая учетная запись используется. Основная часть параметров в этом окне входа в систему главным образом не требует пояснений, а другие параметры мы рассмотрим более подробно.

Поле Server type

Поле Server type позволяет выбрать одну из нескольких подсистем SQL Server, в которую должен войти пользователь (сам сервер базы данных или службы Analysis Services, Report Services и Integration Services). Дело в том, что для различных типов серверов и служб могут использоваться одинаковые имена, поэтому необходимо следить за тем, чтобы в системе действительно осуществлялась регистрация для работы с той службой, которая требуется.

Поле Server name

Как и следовало ожидать, в поле Server name должно быть указано имя сервера SQL Server, к которому должно быть выполнено подключение. На рис. 2.7 показано, что для подключения выбран сервер `SCHWEITZER`. Но другой вариант мог бы предусматривать применение сервера с именем `(local)`. Это не означает, что сервер действительно имеет имя `(local)`, а просто служит указанием на то, что пользователь желает подключиться к применяемому по умолчанию экземпляру SQL Server, эксплуатируемому на том же компьютере, за которым работает пользователь; при этом не учитывается то, какое имя имеет сам компьютер. Выбрав имя `(local)`, можно не

только автоматически задать информацию о том, какой сервер (и экземпляр) требуется использовать, но и сообщить, как должно быть обеспечено взаимодействие клиентской и серверной программ. Вместо (local) можно также использовать в качестве сокращения точку (.).

> В современной версии SQL Server предусмотрена возможность эксплуатировать одновременно несколько экземпляров базы данных SQL Server. При этом просто осуществляется загрузка в память нескольких экземпляров машины SQL Server, работающих независимо друг от друга.

Следует отметить, что экземпляр сервера, применяемый по умолчанию, должен иметь имя, совпадающее с именем компьютера в сети. Безусловно, предусмотрены способы модификации имени сервера после его инсталляции, но эти способы в лучшем случае связаны с возникновением определенных проблем, а в худшем приводят к полному нарушению работы сервера. Дополнительные экземпляры SQL Server могут получать такие же имена, как и имя экземпляра, применяемого по умолчанию (во многих примерах настоящей книги используются имена SCHWEITZER и ARISTOTLE), за которым следует знак доллара и имя самого экземпляра, например ARISTOTLE$POMPEII.

Если выбрано имя (local), то в системе используется библиотека NetLib типа Shared Memory, независимо от того, какая библиотека NetLib была выбрана для обеспечения взаимодействия с другими серверами на предыдущих этапах настройки конфигурации. Подход, в котором применяются локальные серверы, имеет свои преимущества и недостатки. Недостатком этого подхода является то, что при его использовании возможности контроля над функционированием сервера немного уменьшаются (в СУБД SQL Server для подключения многих служебных программ всегда используется библиотека Shared Memory, поэтому возможность выбрать какой-то другой вариант подключения отсутствует). А преимуществом данного подхода является то, что пользователю не приходится запоминать имя сервера, на котором он работает, кроме того, для работы на том же компьютере применяется вариант, обеспечивающий наивысшую производительность. Если же для подключения к серверу, эксплуатируемому на локальном компьютере, используется его действительное имя, то обмен данными происходит через стеки сетевых протоколов и создаются такие же издержки, с которыми приходится сталкиваться в процессе взаимодействия с базой данных, находящейся на другом компьютере, даже несмотря на то, что клиент и сервер функционируют на одном и том же компьютере.

Теперь рассмотрим вопрос о том, как освежить в памяти имя сервера, к которому было выполнено подключение. Для этого достаточно щелкнуть на кнопке со стрелкой, направленной вниз, которая находится справа от поля Server name, чтобы получить список серверов, к которым недавно было выполнено подключение. При прокрутке вниз по этому списку обнаруживается опция <Browse for more...>. Если вы выберете эту опцию, то СУБД SQL Server проведет опрос в сети для определения всех серверов, которые анонсируют свои службы в сети. По существу, анонсирование — это способ, с помощью которого сервер может передать в другие системы информацию о своем местонахождении в сети. Как показано на рис. 2.8, при этом открывается окно Browse for Servers, на котором имеются две вкладки: на одной отображаются локальные серверы (все экземпляры SQL Server, эксплуатируемые в той же системе, за которой работает пользователь), а на другой — иные экземпляры SQL Server в сети.

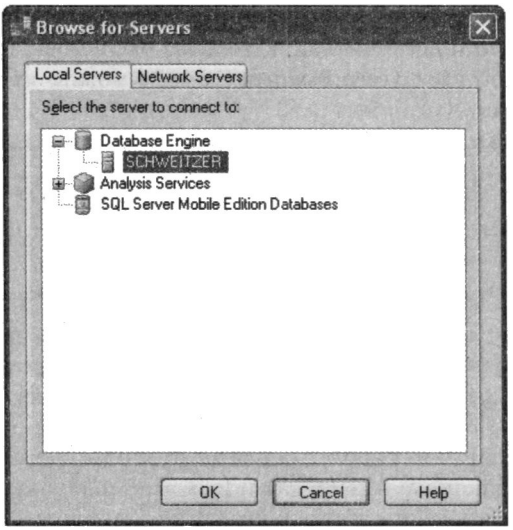


Рис. 2.8. Окно Browse for Servers

Пользователь может выбрать для подключения один из обнаруженных серверов, а затем щелкнуть на кнопке OK.

При использовании для выбора сервера диалогового окна, показанного на рис. 2.8, необходимо быть очень внимательным. Безусловно, этот способ поиска серверов довольно надежный, но следует учитывать, что при некоторых вариантах настройки конфигурации SQL Server широковещательная рассылка информации, с помощью которой анонсируется сервер, становится невозможной. Если настройка конфигурации сервера была выполнена таким образом, то сервер не обнаруживается в списке. Кроме того, в списке не появляются серверы, которые принимают запросы на установление соединения только с помощью библиотеки NetLib типа TCP/IP, и им не соответствует какая-либо запись в службе DNS. В таком случае необходимо заранее знать IP-адрес сервера и обращаться к серверу с его помощью.

Поле Authentication

В поле Authentication может быть выбран один из двух вариантов аутентификации пользователя — **Windows Authentication** (который прежде именовался NT Authentication) и **SQL Server Authentication**. Вариант Windows Authentication всегда остается доступным, независимо от того, как была выполнена настройка конфигурации сервера, т.е. даже в том случае, если он настроен на использование метода аутентификации SQL Server Authentication. Варианты входа в систему с использованием имен пользователей и паролей, которые являются локальными по отношению к серверу SQL Server (и неприменимы в более крупных масштабах сети Windows), становятся приемлемыми для системы, только если специально выбрано значение SQL Server Authentication.

Параметр Windows Authentication

Вариант Windows Authentication полностью соответствует своему названию. Дело в том, что в Windows 2000 и последующих версиях предусмотрена возможность приме-

нять учетные записи пользователей и групп. Учетные записи пользователей Windows отображаются на учетные записи пользователей SQL Server в соответствующих профилях пользователей Windows. При попытке пользователя зарегистрироваться в СУБД SQL Server сведения о пользователе проверяются в домене Windows и отображаются на роли, соответствующие учетной записи, а роли указывают, какие действия разрешается осуществлять пользователю.

Наиболее удобной особенностью данного подхода является то, что для пользователя достаточно иметь только один пароль (а если этот пароль будет изменен в домене Windows, то он изменится и для учетных записей SQL Server). Для того чтобы войти в систему, пользователю не приходится заполнять о себе все сведения; в СУБД просто используется та регистрационная информация, с помощью которой пользователь в текущий момент подключается к системе Windows. Дополнительным преимуществом является то, что администратору не приходится вносить изменения в данные о пользователях в нескольких разных местах. Но этот подход имеет и определенный недостаток, связанный с тем, что процесс отображения учетных записей в определенных обстоятельствах может оказаться сложнее; к тому же заниматься управлением учетными записями пользователей Windows может только администратор базы данных, который одновременно выполняет функции администратора домена.

Параметр SQL Server Authentication

При использовании варианта организации защиты на основе параметра SQL Server Authentication полностью игнорируется вопрос о том, какие права предусмотрены для пользователя в сети, а рассматриваются только права, явно заданные в СУБД SQL Server. В процессе аутентификации вообще не учитывается наличие текущей сетевой регистрационной записи. Вместо этого пользователь задает имя учетной записи и пароль, относящиеся к СУБД SQL Server.

Преимуществом этого варианта организации защиты является то, что администратор данной конкретной СУБД SQL Server не обязан брать на себя функции администратора домена (или даже учитывать то, какое имя имеет пользователь в сети), чтобы предоставить права доступа пользователя к базе данных SQL Server. Кроме того, обычно процесс создания учетной записи пользователя становится немного проще, чем при использовании варианта Windows Authentication. Наконец, вариант SQL Server Authentication позволяет создавать для одного и того же пользователя несколько учетных записей для доступа к разным серверам и службам, предоставляя ему при этом различные права.

Практическое занятие

Создание соединения

Рассмотрим процедуру создания соединения. Если вы впервые подключаетесь к СУБД SQL Server, то выберите в окне входа в систему такие же значения, как показано на рис. 2.7, но с учетом описанных ниже изменений.

1. Выберите в поле Server name значение `(local)`.
2. Выберите вариант SQL Server Authentication.
3. Введите в поле Login имя `sa` (сокращение от System Administrator — системный администратор) и запомните это имя на будущее. Еще один вариант состоит в том, что вы можете войти в систему под именем другого пользователя, при условии, что этот пользователь имеет привилегии системного администратора.

4. Введите пароль учетной записи `sa`, который был задан во время инсталляции СУБД SQL Server. Если для сервера заданы опции, определяющие чувствительность к регистру, то имя учетной записи также необходимо вводить, соблюдая правильный регистр, поэтому обязательно введите это имя строчными буквами.

Если вы подключаетесь к серверу, инсталляцию которого проводил кто-то другой, или если в процессе инсталляции были внесены изменения в значения параметров, применяемые по умолчанию, то необходимо иметь в своем распоряжении регистрационную информацию, которая отражает эти изменения. После щелчка на кнопке OK должно появиться главное окно ввода запросов, показанное на рис. 2.9.

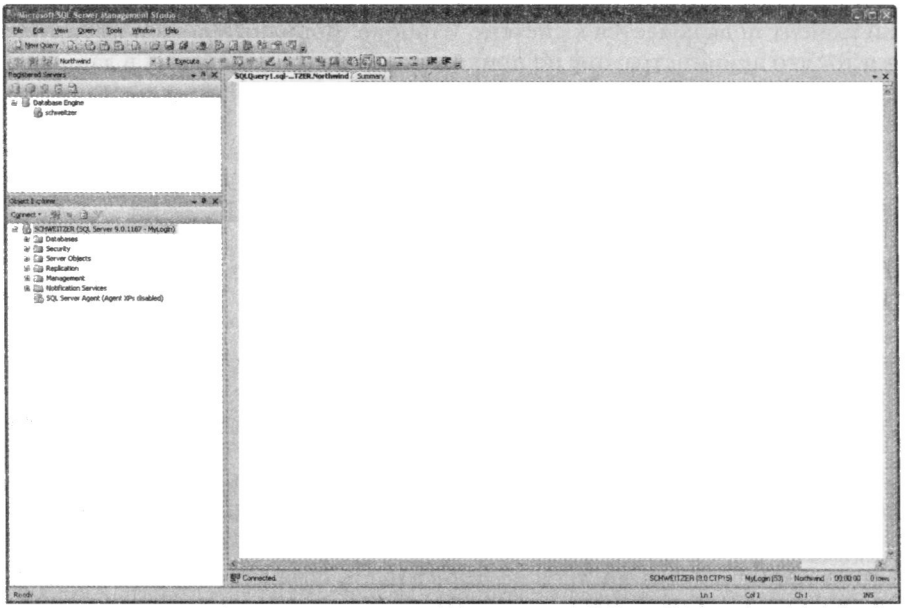


Рис. 2.9. Окно ввода запросов

Сохраняйте в тайне пароль пользователя `sa`. Этот пользователь, а также все прочие пользователи, являющиеся системными администраторами, выполняют функции суперпользователя, имеющего полный доступ ко всем компонентам системы.

Описание полученных результатов

В диалоговом окне Connect to Server собрана вся информация, необходимая для создания соединения. После ввода этой информации сведения о будущем соединении объединяются в единственную строку соединения, которая передается на сервер. После этого осуществляется проверка запроса на установление соединения, и если принимается решение о том, что этот запрос является допустимым, формируется дескриптор соединения и передается в окно ввода запросов. С помощью этого окна пользователь может снова и снова вводить запросы до тех пор, пока соединение не будет разорвано.

Дополнительная информация о том, как происходит создание и форматирование строк соединения, приведена в следующих главах.

В окне, показанном на рис. 2.9, предусмотрена возможность выполнять очень много таких операций (New, Open, Save, Cut, Paste и т.д.), с которыми приходится неоднократно сталкиваться во время работы с другими приложениями Windows, но задано также значительное количество других операций, относящихся только к СУБД SQL Server. Но на данный момент достаточно в основном отметить, что меню в программе Management Studio чувствительны к контексту. Иными словами, состав доступных меню и их содержимое зависит от того, какое окно активизировано в программе Management Studio. Изучая различные части программы Management Studio, обязательно ознакомьтесь со всеми контекстными меню.

Окно ввода запросов

Окно ввода запросов программы Management Studio применяется вместо инструментального средства, которое было выделено в предыдущих версиях в отдельное приложение, называемое **Query Analyzer**. Окно ввода запросов является основным инструментом проведения интерактивных сеансов с любой отдельно взятой СУБД SQL Server. Именно в этом окне можно вызывать на выполнение операторы языка Transact-SQL (T-SQL). Язык T-SQL специально предназначен для использования в СУБД SQL Server. Он является диалектом языка SQL (Structured Query Language) и совместим с версией SQL по стандарту ANSI 92 в минимальной конфигурации. Совместимость в минимальной конфигурации означает, что в СУБД SQL Server поддерживается первый уровень требований, необходимых для классификации программного продукта как совместимого со стандартом ANSI. Практика показывает, что поддержка стандарта ANSI исключительно в минимальной конфигурации предусмотрена в большинстве программных продуктов, относящихся к категории реляционных СУБД.

Сам автор не испытывает особого восторга по поводу появления этой новой версии инструментального средства ввода запросов. Я обнаружил, что из-за большого количества операций, выполняемых в одном инструментальном средстве, пользовательский интерфейс становится громоздким, поэтому затрудняется поиск требуемой информации. Тем не менее следует отметить, что корпорация Microsoft внесла эти изменения в расчете на то, что для начинающих пользователей будет проще использовать средства ввода запросов в составе более крупной программы, Management Studio.

Выполнению запросов, заданных в окне ввода запросов, будет посвящена значительная часть настоящей книги, поэтому рассмотрим это инструментальное средство более внимательно и ознакомимся с его использованием.

Предварительные сведения

До сих пор в этой книге в основном были приведены теоретические сведения, а в связи с изложением данной темы мы можем приступить к выполнению практической работы. Для этого откройте новое окно ввода запросов, щелкнув на кнопке New Query, находящейся в верхнем левом углу окна программы Management Studio, или выбрав команду File⇨New⇨New Query With Current Connection из меню File. После того как откроется окно ввода запросов, появятся меню, в основном идентичные меню программы Query Analyzer, когда эта программа представляла собой отдельное

инструментальное средство. Подробные сведения об этих меню приведены ниже, а сейчас попытаемся ввести свой первый запрос.

Введите следующий код в окне ввода запросов:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

Обратите внимание на то, как изменяется цветовое выделение ключевых слов и конструкций по мере их ввода. Ключевые слова оператора должны быть выделены синим цветом; нераспознаваемые элементы, такие как имена столбцов и таблиц (которые могут быть приняты произвольно для каждой таблицы в каждой базе данных на каждом сервере), — черным цветом; параметры и знаки операций в операторах — красным цветом. Разберитесь в том, как действует цветовое выделение и научитесь им пользоваться. Связанные с цветовым выделением возможности позволяют вылавливать многие программные ошибки еще до того, как будет передан на выполнение сам оператор (и получено результирующее сообщение об ошибке). Пиктограмма на панели инструментов со знаком галочки представляет собой еще один простой элемент отладки, с помощью которого осуществляется оперативный синтаксический анализ запроса еще до осуществления попытки передать соответствующий оператор на выполнение. Это позволяет сразу обнаружить синтаксические ошибки в тексте оператора, не дожидаясь появления сообщения об ошибке, сформированного самой системой. Еще одним средством поиска ошибок может служить отладчик. Более подробная информация по этой теме приведена в главе 12.

Теперь щелкните на кнопке Execute (Выполнить) панели инструментов; эта кнопка обозначена стоящим перед ней красным восклицательным знаком. После этого в окне ввода запросов произойдут изменения, как показано на рис. 2.10.

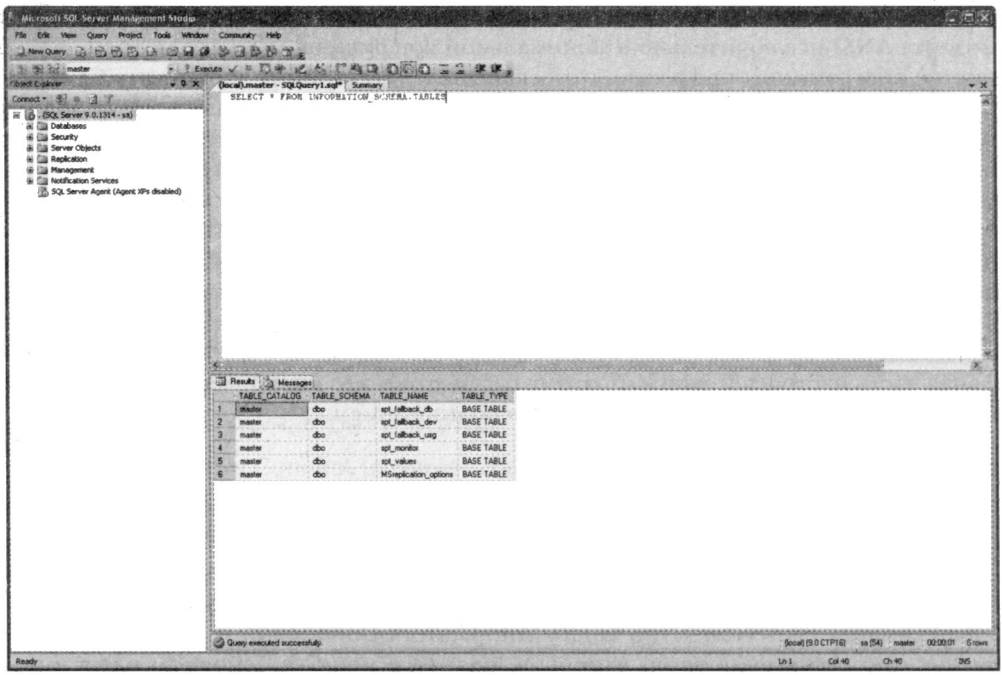


Рис. 2.10. Изменения в окне ввода запросов

Обратите внимание на то, что теперь главное окно автоматически разделено на две области (см. рис. 2.10). В верхней области остался первоначально введенный текст запроса, а в нижней, называемой **областью результатов**, выведены полученные результаты. Кроме того, следует отметить, что в верхней части области результатов имеются вкладки. В дальнейшем, после вызова на выполнение запросов, которые возвращают несколько наборов данных, обнаружится, что каждую из этих совокупностей результатов можно просматривать на отдельных вкладках. Такая возможность является очень удобной, поскольку зачастую невозможно заранее предвидеть, какой объем должен иметь каждый полученный набор данных, или **результирующий набор**.

> Для обозначения набора данных, возвращаемого в результате выполнения некоторой команды, часто используются термины результирующий набор и набор строк. Все приведенные здесь термины могут рассматриваться как взаимозаменяемые.

После этого откорректируем значения одного-двух параметров и рассмотрим, к каким изменениям это приведет. Обратите внимание на то, что над окном ввода запросов имеется панель инструментов, особо отметив ряд, состоящий из трех пиктограмм, которые обведены рамкой на рис. 2.11.

Эти пиктограммы позволяют определять способ оформления выходных данных. В порядке слева направо эти пиктограммы обозначаются как Results in Text, Results in Grid и Results to File. Такие же варианты оформления вывода можно выбрать с помощью меню Query, раскрыв в нем подменю Results To.

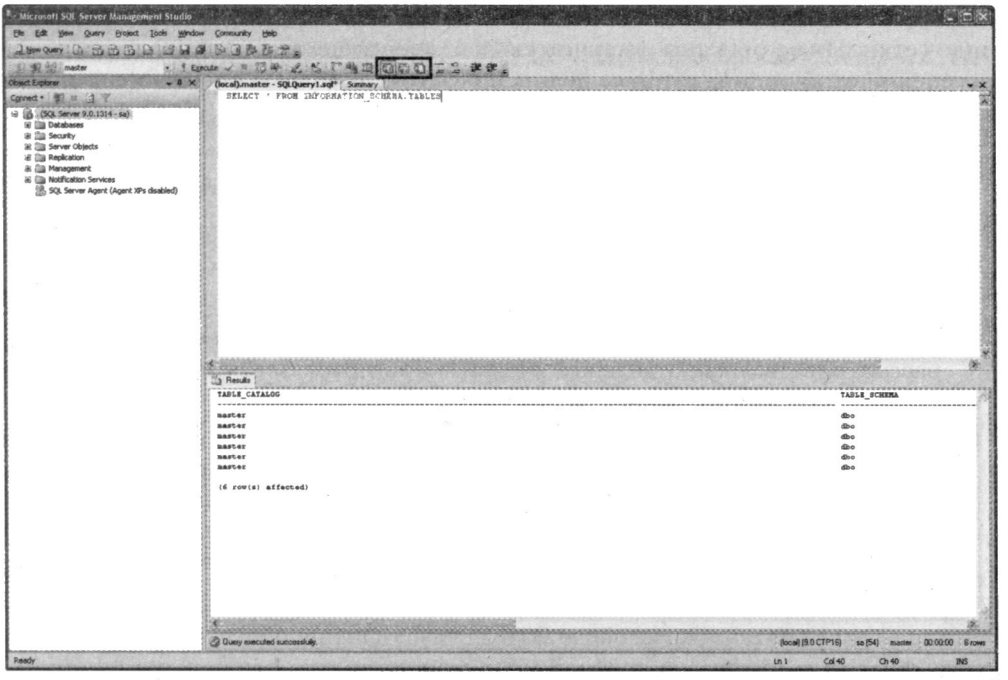


Рис. 2.11. Пиктограммы Results in Text, Results in Grid и Results to File

Опция Results in Text

При использовании опции Results in Text все выходные данные запроса помещаются на одну страницу с текстовыми результатами. Страница может иметь фактически сколь угодно большую длину (ограниченную только объемом доступной памяти в системе).

Прежде чем приступить к дальнейшему изучению этой темы, повторно запустите предыдущий запрос с использованием этой опции и ознакомьтесь с полученными данными. Для этого выберите опцию Results in Text и снова выполните предыдущий запрос, щелкнув на зеленой стрелке (рис. 2.11).

Полученные данные полностью совпадают с теми, что были сформированы перед этим. Единственное отличие состоит в том, что они представлены в другом формате. Автор рекомендует использовать такой метод вывода в ситуациях, описанных ниже.

- При получении единственного результирующего набора, столбцы которого имеют довольно небольшую ширину.
- Если есть необходимость сохранить полученные результаты в единственном текстовом файле.
- Если должно быть получено несколько результирующих наборов, но ожидаемый объем возвращаемых результатов невелик и желательно вывести несколько результирующих наборов на одну страницу, чтобы не приходилось использовать для их просмотра несколько линеек прокрутки.

Опция Results in Grid

При использовании опции Results in Grid данные столбцов и строк располагаются в виде сетки. Ниже описаны дополнительные преимущества, предоставляемые при применении этой опции, которые нельзя получить с помощью опции Results in Text.

- Предоставляется возможность изменять размеры столбца; для этого достаточно провести указатель мыши над правым краем заголовка столбца, а затем щелкнуть и перетащить край столбца, установив его новый размер. А двойной щелчок на правом краю рамки столбца в области заголовка приводит к тому, что выполняется автоматическая подгонка ширины столбца в соответствии с размерами полей с данными.
- Разрешено выбирать сразу несколько ячеек, а затем вырезать их и вставлять в другую сетку (скажем, в окне программы Microsoft Excel), после чего они будут рассматриваться как отдельные ячейки. (При использовании опции Results in Text все вырезанные данные вставляются в одну ячейку.)
- Обеспечивается возможность выбрать только один или два столбца в данных, состоящих из нескольких строк. (Если при использовании опции Results in Text выбирается несколько строк, то во всех промежуточных строках выбирается каждый столбец; с другой стороны, если требуется обозначить подсветкой (выбрать) только часть столбцов, то допускается отмечать только подряд идущие столбцы.)

Автор использует опцию Results in Grid для выполнения почти любой работы, поскольку обычно требуется по крайней мере одна из только что перечисленных возможностей.

Просмотр плана выполнения

После вызова каждого запроса на выполнение в SQL Server осуществляется синтаксический анализ запроса, потом запрос подразделяется на составные части, которые передаются **оптимизатору запросов**. Оптимизатор запросов представляет собой компонент СУБД SQL Server, который определяет наилучший способ выполнения запроса, находя компромисс между такими конфликтующими требованиями, как максимально быстрое получение результатов и оказание минимального отрицательного воздействия на работу других пользователей. Применение опции Show Estimated Execution Plan (Показать предполагаемый план выполнения) позволяет получить графическое представление и дополнительную информацию о том, как СУБД SQL Server планирует выполнение запроса. Аналогичным образом, может быть разрешено применение опции Include Actual Execution Plan (Включить действительный план выполнения). Чаще всего действительный план выполнения совпадает с предполагаемым планом выполнения, но иногда между этими планами выполнения обнаруживаются различия, связанные с тем, что оптимизатором было принято решение внести изменения в план во время прогона запроса, а также обнаружились расхождения между действительной стоимостью выполнения запроса и тем значением стоимости, которое было первоначально принято оптимизатором в качестве предположения.

Ознакомьтесь с тем, как выглядит план запроса на примере рассматриваемого в данной главе простого запроса. Для этого щелкните на пиктограмме Include Actual Execution Plan и снова вызовите запрос на выполнение, как показано на рис. 2.12.

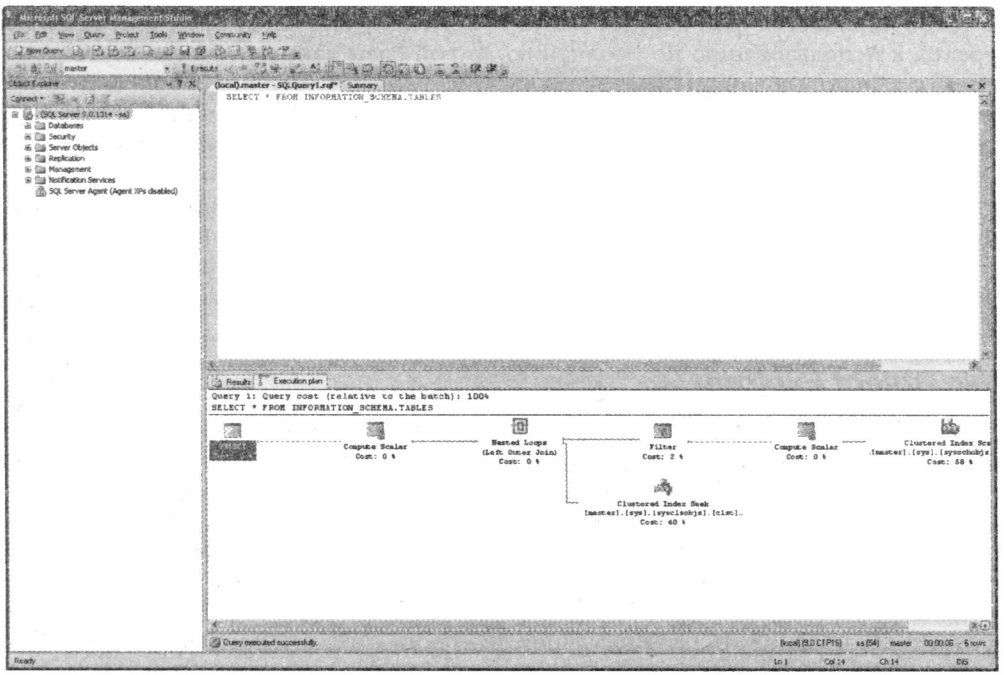


Рис. 2.12. Выполнение запроса с применением опции Include Actual Execution Plan

Обратите внимание на то, что для просмотра плана выполнения необходимо щелкнуть на вкладке Execution Plan, а результаты запроса по-прежнему отображаются в том формате, который для них выбран. Опция Show Estimated Execution Plan позволяет получить такие же выходные данные, как и опция Include Actual Execution Plan, не считая двух указанных ниже различий.

- При использовании опции Show Estimated Execution Plan план предоставляется немедленно, а не после выполнения запроса.
- Опция Show Estimated Execution Plan позволяет ознакомиться с планом, с применением которого действительно будет происходить запуск запроса на выполнение, но вся информация о стоимости является предполагаемой, полученной еще до фактического выполнения запроса. А при использовании опции Include Actual Execution Plan план запроса отражает состояние, достигнутое после физического выполнения запроса, и полученная информация о стоимости является действительной, а не предполагаемой.

Поле со списком баз данных

Наконец, интерес представляет **поле со списком баз данных** программы SQL Server Management Studio. Коротко можно отметить, что именно это поле позволяет выбрать применяемую по умолчанию базу данных, данные которой будут использоваться при выполнении запросов, представленных в текущем окне. Первоначально окно запросов открывается для работы с базой данных, применяемой по умолчанию для учетной записи, в которой зарегистрировался пользователь (для пользователя `sa` таковой является база данных `master`, если только в системе не был выполнен переход к использованию по умолчанию другой базы данных). После этого можно перейти к использованию любой другой базы данных, доступ к которой разрешен для текущей учетной записи. А поскольку в данном примере применяется идентификатор пользователя `sa`, то в поле со списком баз данных должны быть представлены все базы данных, относящиеся к текущему серверу.

Выберите вместо базы данных, используемой в настоящее время, базу данных `AdventureWorks` и снова вызовите запрос на выполнение (рис. 2.13).

Вполне очевидно, что данные изменились в соответствии с тем, что теперь они представляют данные из базы данных, применяемой в повторно выполненном запросе (см. рис. 2.13).

Окно Object Explorer

Окно Object Explorer представляет собой небольшое, но очень полезное инструментальное средство, позволяющее переходить от одного объекта базы данных к другому, просматривать имена объектов и даже осуществлять такие действия, как прогон сценариев и просмотр основополагающих данных.

На рис. 2.14 показан фрагмент окна Object Explorer, в котором развернут узел с обозначением базы данных `AdventureWorks`. Этот узел базы данных может быть развернут дополнительно, вплоть до получения листингов таблиц. Предусмотрена даже возможность продолжать развертывание вплоть до того, чтобы можно было раскрывать отдельные столбцы таблиц и проверять соответствующие им типы данных и подобные свойства. Таким образом, окно Object Explorer — очень удобное инструментальное средство просмотра баз данных.

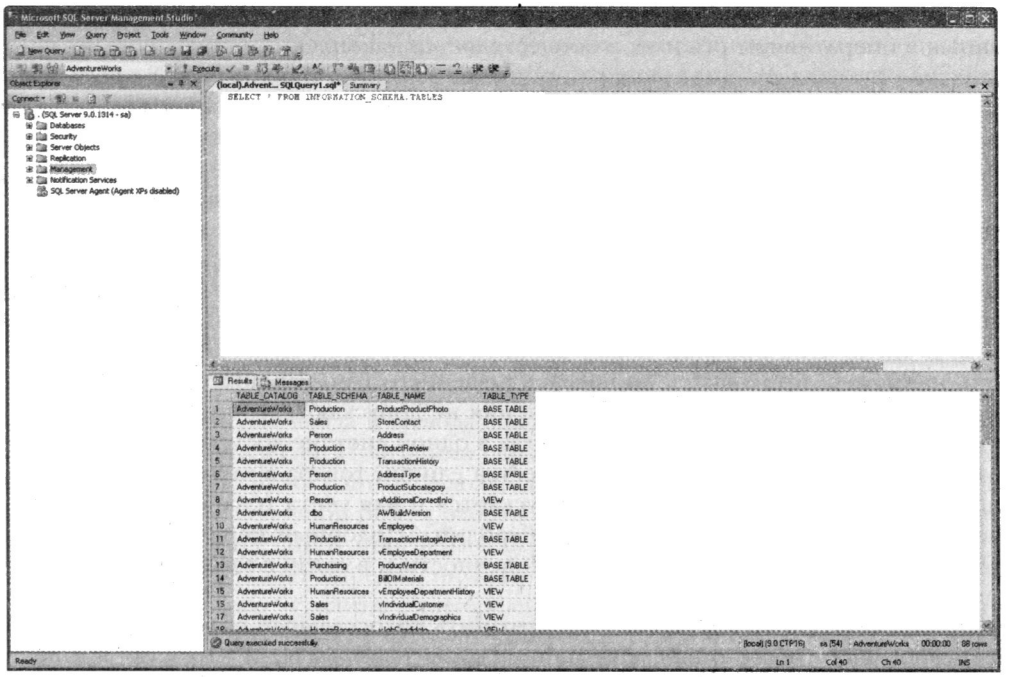


Рис. 2.13. Выполнение запроса в базе данных `AdventureWorks`

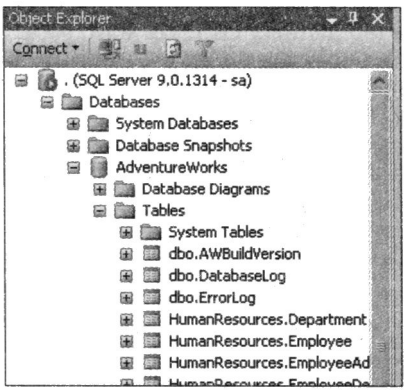


Рис. 2.14. Фрагмент окна Object Explorer

Службы SSIS

Важность служб SSIS (SQL Server Integration Services), которые прежде носили название служб DTS (Data Transformation Services), буквально невозможно переоценить. Автор всегда испытывает восхищение, пользуясь этим средством СУБД SQL Server. А для того чтобы читатель понял, какие возможности открываются лично для него, отмечу, что я за последние годы разработал с помощью службы SSIS целый ряд проектов систем поддержки принятия решений (Decision Support System – DSS).

В системах поддержки принятия решений обычно не происходит ввод и вывод данных в оперативном режиме, а осуществляется накопление данных в целях предоставления руководителям помощи в принятии решений. В проекте DSS предусматривается сбор данных из разнообразных источников и перекачка этих данных в централизованную базу данных для дальнейшего использования при централизованной подготовке отчетов.

С началом разработки любых проектов системы такого типа вскоре обнаруживается чрезвычайно стремительное возрастание затрат на реализацию этих проектов, поскольку в них приходится сталкиваться с тем фактом, что в различных системах данные, по существу являющиеся одинаковыми, могут обозначаться разными идентификаторами. При этом приходится сталкиваться с бесконечным множеством проблем. В число этих проблем может входить обеспечение целостности данных (для чего, например, приходится принимать решение о том, как поступить, если в поле содержится NULL-значение, а эти значения не допускаются) или преодоление различий в бизнес-правилах (например, когда в одной системе допускается выдача товаров в кредит путем обозначения количества единиц товара в заказе отрицательным числом, а в другой системе это не допускается и для учета товаров, отпускаемых в кредит, применяется отдельный набор таблиц). Подобный перечень реализуемых требований может расширяться до бесконечности и в связи с этим существенно возрастают расходы.

Применение служб SSIS позволяет устранить колоссальный объем работы по программированию (обычно на каком-либо из клиентских языков), которую, как правило, приходилось выполнять в целях осуществления необходимых преобразований. А при использовании служб SSIS эта работа, по крайней мере, значительно упрощается. Службы SSIS позволяют получить данные из любого источника данных, для которого предусмотрен провайдер данных OLE DB или .NET, а затем ввести эти данные в таблицу SQL Server.

Следует иметь в виду, что предусмотрен специальный провайдер OLE DB для интерфейса ODBC. Этот провайдер позволяет обеспечить непосредственное отображение средств доступа OLE DB на драйвер ODBC. Это означает, что все данные, к которым можно получить доступ с помощью интерфейса ODBC, становятся также доступными с использованием провайдера OLE DB (и, следовательно, служб SSIS).

Раз уж речь зашла об этой теме, необходимо также отметить, что службы SSIS, хотя и входят в состав программного обеспечения SQL Server, могут работать с любым источником OLE DB и отправлять данные по любому назначению OLE DB. Это означает, что при перекачке данных вообще можно обойтись без участия в этом процессе самой СУБД SQL Server. Например, с помощью служб SSIS можно обеспечить передачу данных из базы данных Oracle в таблицы Excel или даже из СУБД DB/2 в СУБД MySQL.

В процессе передачи данных к этим данным могут быть также применены так называемые **преобразования**. Преобразование по сути представляет собой модификацию данных в соответствии с каким-либо алгоритмическим правилом (правилами). Модификация может быть настолько простой, как изменение имени столбца, или настолько сложной, как проверка целостности данных и применение в случае необходимости соответствующих правил для внесения изменений в данные. Чтобы ознакомиться с тем, как может выполняться преобразование, достаточно упомянуть

приведенный выше пример, в котором данные извлекаются из столбца, для которого разрешено применение NULL-значений, после чего эти значения переносятся в столбец другой таблицы, в котором не допускается наличие NULL-значений. Службы SSIS позволяют автоматически подставлять вместо NULL-значений какие-то другие значения, выбранные в процессе передачи (в частности, для числового поля может быть применена подстановка нуля, а для символьного поля — какой-то строки наподобие "unknown").

Программа bcp

Программа bcp (Bulk Copy Program) относится к той же категории программного обеспечения, что и службы SSIS, но постепенно заменяется последними.

Программа bcp — это утилита с интерфейсом командной строки, предназначенная исключительно для ввода и вывода значительных объемов отформатированных данных в СУБД SQL Server. Программа bcp применялась как основное средство массовой загрузки и выгрузки данных задолго до того, как были созданы службы SSIS, а в настоящее время основной объем работы по осуществлению операций импорта и экспорта данных передан от программы bcp к службам SSIS. Тем не менее программа bcp все еще сохраняет определенную привлекательность для тех пользователей, которые предпочитают утилиты с интерфейсом командной строки, поэтому количество инсталляций SQL Server, в которых программа bcp широко используется для быстрого перемещения данных, все еще чрезвычайно велико.

Программа SQL Server Profiler

Программа SQL Server Profiler позволяет получить информацию о том, что происходит в серверном программном обеспечении, даже в таких безвыходных ситуациях, когда не остается больше никаких других возможностей. Безусловно, разработчику (или даже администратору базы данных) не приходится использовать эту программу в своей повседневной работе, но она является чрезвычайно мощной и позволяет найти выход из положения, когда больше ничто не может вам помочь.

Кратко можно отметить, что программа **SQL Server Profiler** представляет собой инструментальное средство текущего контроля в реальном времени. Программа Performance Monitor позволяет отслеживать все, что происходит на самом верхнем уровне системы (на уровне определения конфигурации системы), а программа SQL Server Profiler отслеживает конкретные события. В этом состоит и преимущество, и недостаток программы. В зависимости от того, какие параметры применялись для настройки средств трассировки, программа SQL Server Profiler позволяет получить конкретные сведения о синтаксической структуре буквально каждого оператора, выполняемого на сервере. Узнав об этом, достаточно представить себе, что осуществляется настройка производительности в системе с одной тысячей пользователей. В таком случае для распечатки всех операторов, выполняемых столь большим количеством пользователей даже в течение одной или двух минут, потребовались бы буквально горы бумаги. К счастью, в программе SQL Server Profiler предусмотрен обширный массив фильтров, которые позволяют сузить область поиска и сосредоточиться на отслеживании фактов проявления более определенных проблем, например, долго выполняющихся запросов. Еще одна возможность состоит в том, что программа SQL

Server Profiler позволяет точно узнать текст запроса, вызываемого на выполнение из хранимой процедуры, а это очень удобно, если в процедуре применяется ряд условных операторов, в результате чего выполняются разные действия при различных обстоятельствах.

Программа sqlcmd

Программа sqlcmd не представлена в группе программ SQL Server. В действительности многие разработчики даже не подозревают о существовании этой утилиты или о том, что когда-то существовали такие программы, как osql и isql, впервые появившиеся в еще более ранних версиях SQL Server. Дело в том, что программа sqlcmd имеет интерфейс командной строки, а не графический интерфейс Windows.

Программа sqlcmd — это инструментальное средство, которое позволяет выполнять пакетные файлы, состоящие из операторов SQL. До выхода версии SQL Server 7.0 и появления служб DTS (которые теперь заменены службами SSIS) программа sqlcmd часто использовалась вместе с программой bcp для обеспечения импорта данных из внешних систем. Но в последнее время для загрузки и выгрузки данных все чаще применяются службы SSIS, поэтому данная область применения программы sqlcmd постепенно сходит на нет. Тем не менее иногда возникает необходимость организовать выполнение ряда операторов SQL с помощью утилиты с интерфейсом командной строки, и такую возможность предоставляет программа sqlcmd.

Программа sqlcmd может оказаться очень удобной, особенно если под ее управлением необходимо выполнить сценарии, оформленные в виде файлов. Однако следует учитывать, что некоторые инструментальные средства позволяют достичь тех же результатов, что и с использованием программы sqlcmd, но значительно эффективнее. К тому же при этом все необходимые действия осуществляются с помощью пользовательского интерфейса, который лучше всего совместим с другими программами, применяемыми во время работы с СУБД SQL Server.

Необходимо отметить, что в первых версиях SQL Server утилита с интерфейсом командной строки, предназначенная для выполнения пакетных файлов, носила имя isql. В версиях SQL Server 2000 и 7.0 для этой цели применялась утилита osql. А утилита sqlcmd введена в действие в версии SQL Server 2003.

Резюме

Основная часть инструментальных средств, описанных в настоящей главе, не предназначена для повседневного использования. В действительности разработчики регулярно применяют только такие программы, как SQL Server Management Studio, но должны иметь определенное представление о том, для чего предназначена каждая утилита, поскольку все служебные программы в какой-то момент могут оказать существенную помощь. Дополнительные сведения об инструментальных средствах, представленных в этой главе, можно также найти в других главах данной книги.

Необходимо учитывать, что в меню Start нет ярлыков для некоторых других доступных утилит (таких как инструментальные средства обеспечения связи по сети, средства диагностирования сервера и утилиты сопровождения), поскольку они главным образом предназначены для выполнения задач администрирования.

3

Основные операторы языка T-SQL

В предыдущих главах были представлены очень важные, но не столь интересные сведения. Обычно сложнее всего заниматься изучением основных объектов и инструментальных средств. Но, к сожалению, прежде чем приступить к строительству дома, необходимо заложить фундамент. Однако теперь мы можем вздохнуть с облегчением, поскольку фундамент дальнейшего изучения этой книги уже готов. Продолжая эту аналогию со строительством дома, отметим, что в дальнейшем изложении речь пойдет о том, как воспользоваться благами цивилизации, предусмотренными в современном доме, и только потом будет описано, как они устроены. А применительно к эксплуатации баз данных мы вначале рассмотрим, как осуществляется доступ к данным, и лишь затем изложение будет в большей степени посвящено тому, каковы наилучшие способы хранения самих данных.

В настоящей главе рассматриваются наиболее фундаментальные операторы **языка Transact-SQL** (или сокращенно **T-SQL**). Язык T-SQL – это собственный диалект **языка структурированных запросов** (или сокращенно **SQL**), применяемый в СУБД SQL Server. При подготовке данного выпуска СУБД SQL Server язык T-SQL был в значительной степени доработан и в него добавлены многие новые программные конструкции. Кроме всего прочего, он был преобразован в язык, совместимый с общей средой выполнения (Common Language Runtime – CLR); короче говоря, начиная с этого выпуска T-SQL стал одним из языков .NET. Кроме того, благодаря выпуску SQL Server 2005 появилась возможность использовать для доступа к базе данных любой язык .NET, но в данной книге речь пойдет в основном о языке T-SQL, поскольку T-SQL всегда будет оставаться ведущим языком, позволяющим выполнять любые операции в СУБД SQL Server.

В настоящей главе рассмотрены следующие операторы T-SQL:

- SELECT.
- INSERT.
- UPDATE.
- DELETE.

Эти четыре оператора представляют собой альфу и омегу языка T-SQL. Ниже мы рассмотрим целый ряд других операторов, но именно эти четыре операторы составляют основу **языка манипулирования данными (Data Manipulation Language — DML)**, входящего в состав T-SQL. Вообще говоря, в процессе работы с базой данных гораздо чаще приходится пользоваться командами, предназначенными для манипулирования данными (т.е. чтения и модификации), чем командами других типов (например, предназначенными для предоставления прав пользователям или создания таблиц), поэтому читатель даже не заметит, как эти четыре оператора станут для него хорошо знакомыми.

Кроме того, в языке SQL предусмотрено много операций и ключевых слов, позволяющих уточнять назначение запросов. Часть наиболее распространенных конструкций этого типа также рассматривается в данной главе.

Язык T-SQL предназначен исключительно для работы с СУБД SQL Server, но основная часть применяемых в нем операторов имеет более широкое использование. Язык T-SQL совместим со стандартом ANSI SQL-92 на начальном уровне. Это означает, что в определенном объеме T-SQL совместим с очень широким открытым стандартом. Поэтому основной объем сведений о языке SQL, полученный при изучении данной книги, может непосредственно применяться для работы с другими серверами баз данных с поддержкой SQL, такими как Sybase (отметим, что много лет тому назад в Sybase совместно использовалась общая база кода с SQL Server), Oracle, DB2 и MySQL. Но следует учитывать, что в каждой реляционной СУБД применяются различные расширения и способы повышения производительности, дополняющие указанный стандарт ANSI и выходящие за его рамки. Автор будет подчеркивать различия между способами осуществления действий, предусмотренными и не предусмотренными в стандарте ANSI, когда это будет уместно. В некоторых случаях выбор того или иного способа становится равносильным достижению компромисса между производительностью и переносимостью в другие системы реляционных СУБД. Тем не менее программные средства, предусмотренные стандартом ANSI, обеспечивают не менее высокое быстродействие по сравнению с другими вариантами организации работы. В подобных случаях выбор должен быть очевидным — поддерживать совместимость с указанным стандартом ANSI.

Исходные сведения об использовании основного оператора SELECT

Те читатели, которые еще до сих пор не использовали язык SQL или не чувствуют уверенности в том, что действительно поняли суть этого языка, должны теперь напрячь все свое внимание! Оператор SELECT и применяемые в нем структуры составляют львиную долю всех команд, выполняемых в процессе работы с СУБД SQL Server. Рассмотрим основные синтаксические правила составления операторов SELECT:

```
SELECT <column list>
[FROM <source table(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML {RAW|AUTO|EXPLICIT|PATH [(<element>)]}[, XMLDATA][, ELEMENTS][,
BINARY base 64]]
[OPTION (<query hint>, [, ...n])]
```

Очевидно, что это синтаксическое определение содержит большой объем информации, поэтому мы будем рассматривать его по частям.

Оператор SELECT и конструкция FROM

Основой всего оператора, который сообщает СУБД SQL Server, какое действие она должна выполнить, является так называемый "глагол", в данном случае SELECT. Применение в операторе ключевого слова SELECT указывает на то, что должно быть выполнено только чтение информации, а не ее модификация. Информация, подлежащая выборке, обозначается с помощью выражения или списка столбцов, которые непосредственно следуют за ключевым словом SELECT (вскоре станет ясно, что под этим подразумевается).

Затем необходимо дать некоторые уточнения, в частности, касающиеся того, откуда должны быть получены данные. Для указания имени таблицы (или таблиц), являющейся источником получения данных, служит конструкция FROM. Сказанного выше достаточно для того, чтобы мы могли приступить к созданию простого оператора SELECT. Запустите программу SQL Server Management Studio и еще раз обратите внимание на оператор SELECT, который рассматривался в предыдущей главе:

```
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

Уточним, какая информация здесь запрашивается. Прежде всего серверу передан запрос на выборку информации, SELECT; такой запрос можно также рассматривать как требование вывести информацию на внешнее устройство. Символ * на первый взгляд может показаться странным, но фактически он действует во многом так же, как и в составе любых других средств программирования, т.е. выполняет роль подстановочного символа. Выражение SELECT * по существу означает, что требуется выборка содержимого всех столбцов таблицы. Следующее далее ключевое слово FROM говорит о том, что указания, касающиеся того, какие элементы данных должны быть выведены на внешнее устройство, закончены, и теперь речь пойдет о том, каковым является предполагаемый источник получения информации. В данном случае указано, что источником является таблица INFORMATION_SCHEMA.TABLES.

> INFORMATION_SCHEMA — это специальный путь доступа, который используется для получения метаданных о базах данных, развернутых в системе, и их содержимом. Путь доступа INFORMATION_SCHEMA состоит из нескольких частей (которые могут быть заданы после точки), таких как INFORMATION_SCHEMA.SCHEMATA или INFORMATION_SCHEMA.VIEWS. Эти специальные пути доступа к метаданным о системе были предусмотрены для того, чтобы исключить необходимость в использовании так называемых "системных таблиц".

Практическое занятие Проверка работы введенного оператора `SELECT`

Проведем с использованием указанного оператора некоторые дополнительные действия. Измените текущую базу данных так, чтобы ею стала база данных `AdventureWorks`. Напомним, что для этого достаточно выбрать элемент `AdventureWorks` из поля со списком, находящегося на панели инструментов в верхней части окна `Query` в программе Management Studio (рис. 3.1).

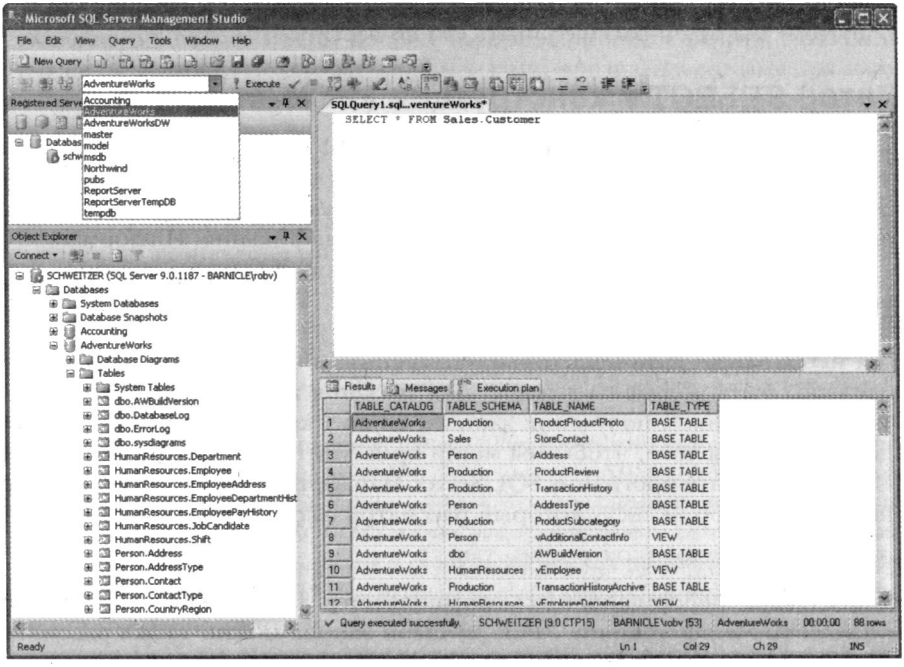


Рис. 3.1. Окно `Query` программы Management Studio

Если вы столкнетесь с затруднениями при поиске поля со списком, в котором перечислены различные базы данных, попытайтесь один раз щелкнуть в окне Query. *Панели инструментов программы SQL Server Management Studio являются контекстно-зависимыми. Это означает, что их содержимое изменяется с учетом тех действий, которые в текущий момент выполняет пользователь. Если, например, окно* Query *не активизировано, то может оказаться, что в нем подготовлен к работе другой набор панелей инструментов (тот, который в большей степени подходит для выполнения какой-то другой задачи). Но как только окно* Query *станет активизированным, оно должно переключиться на тот набор панелей инструментов, который является более подходящим для работы с запросами.*

Теперь, после того как выбрана база данных `AdventureWorks`, приступим к просмотру некоторых реальных данных из этой базы данных. Введите следующий запрос:

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

После того как этот запрос будет введен в окне Query, достаточно щелкнуть на кнопке Execute панели инструментов и понаблюдать за тем, как СУБД SQL Server выдает требуемые результаты. Данный запрос выводит каждую строку данных со всеми столбцами из таблицы `Sales.Customer` текущей базы данных (в данном случае `AdventureWorks`). Если перед запуском на выполнение этого запроса вы не изменили ни одного параметра настройки в своей системе и не модифицировали сами данные в базе данных `AdventureWorks`, то после щелчка на вкладке Messages должны увидеть следующую информацию:

```
(19185 row(s) affected)
```

В случае оператора `SELECT` показанное здесь число обозначает количество строк, возвращенных запросом.

Описание полученных результатов

Рассмотрим некоторые особенности оператора `SELECT`. Прежде всего следует отметить, что автор выделил ключевые слова `SELECT` и `FROM` прописными буквами. Но в СУБД SQL Server такое требование не является обязательным; эти ключевые слова вполне можно ввести как `SeLeCt` и `frOM`, после чего они будут действовать столь же успешно. Таким образом, обозначение операторов с помощью прописных букв используется исключительно для соблюдения определенного соглашения и обеспечения удобства чтения. Читатель сможет сам убедиться в том, что многие программисты, работающие на языке SQL, придерживаются соглашения о выделении прописными буквами всех команд и ключевых слов, а для ввода имен таблиц, столбцов и отличных от констант переменных применяют смешанный регистр букв. Разработчик может выбрать для себя такой стандарт или быть вынужденным применять другие стандарты, но следует всегда придерживаться одного важного правила — неизменно соблюдать одно и то же соглашение, касающееся использования прописных и строчных букв.

Автор снова хочет вскочить на любимого конька — заняться критикой неистребимых недостатков. Нет ничего более раздражающего для постороннего пользователя, который вынужден читать чужой код или запоминать придуманные другими имена таблиц, чем отсутствие единообразия. Если разрабатываемый код предназначен для чтения другими программистами или, что еще более важно, имена создаваемых столбцов и таблиц должны использовать другие разработчики, то от этих людей не должны требоваться значительные усилия для понимания принятого соглашения об именовании. Важно, чтобы им было достаточно воспользоваться опытом работы с теми частями созданного приложения, с которыми они уже знакомы. Почти любая база данных, с которой приходилось работать автору, хотя бы в определенной степени была лишена того достоинства, которого так легко достичь, — единообразия. Покончите с этой тенденцией — соблюдайте требования к единообразию.

Ввод оператора `SELECT` равносилен передаче программному обеспечению окна Query сообщения о том, что мы намереваемся сделать, а символ * указывает, какие данные нам требуются (напомним, что символ * эквивалентен указанию "все столбцы"). За этими обозначениями следует ключевое слово `FROM`.

Конструкция `FROM` выполняет буквально то, что означает ключевое слово `FROM` (Из). Иными словами, эта конструкция определяет место, из которого должны быть получены данные. Непосредственно за ключевым словом `FROM` должны следовать имена одной или нескольких таблиц. В рассматриваемом запросе все данные должны поступить из таблицы `Customer`.

Теперь попытаемся получить немного более конкретную информацию. Предположим, что нам требуется получить список всех заказчиков, указанных по фамилиям:

```
SELECT LastName FROM Person.Contact
```

Полученные результаты будут выглядеть примерно так:

```
Achong
Abel
Abercrombie
...
He
Zheng
Hu
```

Обратите внимание на то, что в целях сокращения строки, находящиеся в середине, удалены, поскольку данный запрос приводит к получению 19 972 строк. В данном случае было решено получить список фамилий всех заказчиков, поэтому и была применена конструкция, предназначенная для выборки всех этих данных.

*Многие разработчики, использующие язык SQL, стремятся создавать как можно более краткие запросы и всегда предусматривают выборку всех столбцов, используя символ * в качестве критерия выборки. Это — еще одна дурная привычка, которую нельзя усваивать. Безусловно, если ввести символ * вместо требуемых имен столбцов, то не понадобится несколько раз нажимать клавиши, но вместе с тем объем данных, поступающих из базы данных, становится больше, чем действительно требуется. Кроме того, на СУБД SQL Server возлагается дополнительная работа, связанная с необходимостью уточнять количество столбцов, подразумеваемых под символом *, а также определять, какими именно являются эти столбцы. Часто после перехода от конкретного указания столбцов к использованию символа * разработчик с удивлением обнаруживает, насколько снизилась производительность приложения и увеличился объем данных, передаваемых по сети. Короче говоря, следует всегда придерживаться хорошего правила — осуществлять выборку только требуемой информации, т.е. той информации, которая действительно требуется, — ни больше ни меньше.*

Ниже приведен еще один простой запрос. Любопытно, к каким результатам он приведет.

```
SELECT Name FROM Production.Product
```

И в этом случае подразумевается, что данные, предусмотренные в этой базе данных, используемой в качестве образца, не были модифицированы. В ответ СУБД SQL Server должна возвратить список из 504 различных товаров, информация о которых имеется в базе данных AdventureWorks:

```
Name
--------------------------------------
Adjustable Race
Bearing Ball
BB Ball Bearing
...
...
Road-750 Black, 44
Road-750 Black, 48
Road-750 Black, 52
```

Столбцы, заданные непосредственно после ключевого слова SELECT, называются **списком выборки**. Короче говоря, список выборки состоит из столбцов, которые, согласно запросу, должны участвовать в формировании выходных данных.

Конструкция WHERE

Итак, мы сумели успешно разобраться еще в одном сложном вопросе. Теперь перейдем к изучению конструкции WHERE, которая позволяет налагать условия, определяющие, какую именно информацию должен возвратить запрос. До сих пор не устанавливались какие-либо ограничения, касающиеся объема возвращаемой информации; иными словами, в условиях запроса подразумевалось, что в итоговые результаты должны быть включены все строки указанной таблицы. Подобные неограниченные запросы являются очень удобными для заполнения таких структур представления данных, как списки и поля со списками, а также в других сценариях организации работы, когда предпринимаются попытки подготовить **листинг доменов**.

> В контексте данного изложения не следует путать простой домен с доменом Windows. Листинг доменов — это список исключительных вариантов выбора. Например, если требуется предоставить в программе информацию о каком-то штате США, то можно предусмотреть использование списка, который ограничивает перечень вариантов только пятьюдесятью штатами. Таким образом, всегда можно будет гарантировать, что выбранный вариант окажется действительным. С этой концепцией доменов мы ознакомимся более подробно, когда речь пойдет о проектировании баз данных.

Теперь мы попытаемся выполнить поиск более конкретной информации. Допустим, что требуется не список с наименованиями товаров, а информация о конкретном товаре. Прежде всего попробуйте самостоятельно подготовить запрос, который возвращает наименование, код товара и информацию о том, каковы условия возобновления запасов этого товара применительно к товару с идентификатором ProductID, равным 356.

Разобьем эту задачу на несколько подзадач и приступим к поэтапному составлению запроса. Прежде всего нам предстоит запросить из базы данных хранимую в ней информацию, поэтому очевидно, что должен использоваться оператор SELECT. В формулировке требований к получаемой информации указано, что нам требуется наименование товара, код товара и информация об условиях возобновления запасов товара, поэтому необходимо узнать имена столбцов, в которых хранятся эти фрагменты информации. Кроме того, необходимо узнать, из какой таблицы или таблиц можно осуществить выборку данных этих столбцов.

Теперь рассмотрим доступные нам таблицы. Поскольку мы уже однажды использовали таблицу Production.Product, то нам известно, что она существует (ниже в этой главе будет показано, как узнать имена доступных таблиц, если они еще не известны). Таблица Production.Product имеет несколько столбцов. Для быстрого ознакомления с имеющимся выбором столбцов можно изучить дерево Object Explorer таблицы Production.Product с помощью программы Management Studio. Чтобы открыть нужное окно в программе Management Studio, щелкните на элементе Tables, находящимся под обозначением базы данных AdventureWorks. В результате этого развернутся узлы Production.Product и Columns. Как показано на рис. 3.2, появится список всех столбцов с обозначением типов хранимых в них данных и с указанием возможности представлять с их помощью неопределенные данные (nullability).

Ниже в этой главе будут описаны и некоторые другие методы поиска необходимой информации.

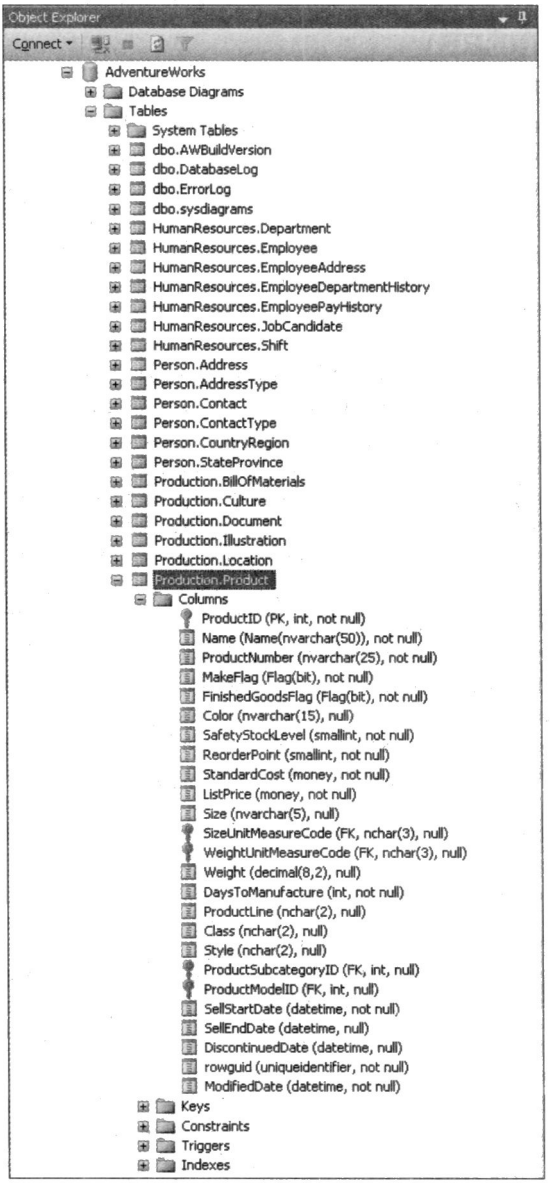


Рис. 3.2. Список столбцов таблицы `Production.Product`

В этом списке не удается найти столбец `product name` (наименование товара), но имеется такой столбец, который, по-видимому, нам как раз и требуется — `Name` (не правда ли, весьма оригинальное имя?). Другие два столбца можно найти гораздо проще, если не считать того, что в их именах отсутствует пробел между двумя словами.

Итак, источником данных, из которого должна быть получена информация, указанным в конструкции FROM, является таблица Products, а конкретными столбцами, в которых находится требуемая информация, являются столбцы Name (Наименование), ProductNumber (Код товара) и ReorderPoint (Условие возобновления запасов):

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
```

Но данный запрос все еще не может использоваться для получения только требуемых результатов, поскольку он по-прежнему возвращает слишком много информации. Вызвав его на выполнение, можно убедиться в том, что этот запрос снова возвращает все записи из таблицы, а не только ту, которая необходима.

Такое положение дел вполне бы нас устроило, если бы в таблице было только несколько записей и нам требовалось быстро с ними ознакомиться. В конечном итоге просмотр небольшого списка — это всегда простая задача. Но, к сожалению, подобная ситуация возникает довольно редко. В любой достаточно развитой системе небольшое количество записей может находиться лишь в незначительной части всех таблиц. А задача просмотра с помощью прокрутки десяти тысяч записей вряд ли покажется кому-то привлекательной. Такое положение дел еще в большей степени усугубляется, если количество записей составляет сотни тысяч или миллионы. Даже если вы готовы просмотреть все записи в поиске нужной информации, затраты времени на получение результатов по мере увеличения количества записей существенно возрастают. Наконец, что делать, если такая большая таблица используется в составе приложения, а в процессе работы требуется быстро получить не все данные подряд, а только нужные результаты?

Поэтому нам требуется такая операция проверки условия, которая позволит свести весь объем результатов запроса лишь к одному идентификатору товара, равному 356. Именно для этой цели и предназначена конструкция WHERE. Конструкция WHERE непосредственно следует за конструкцией FROM и определяет, каким условиям должна соответствовать запись, чтобы рассматривать ее как подлежащую возврату. Для данного запроса требуется, чтобы значение ProductID было равно 356, поэтому окончательно сформулируем рассматриваемый запрос следующим образом:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
WHERE ProductID = 356
```

Вызовите на выполнение этот запрос применительно к базе данных AdventureWorks. Должны быть получены такие результаты:

```
Name                     ProductNumber          ReorderPoint
-----------------------  ---------------------  ------------
LL Grip Tape             GT-0820                600

(1 row(s) affected)
```

На сей раз было получено именно то, что требовалось. Кроме того, этот запрос был выполнен намного быстрее по сравнению с первым запросом.

Все операции, которые могут использоваться в конструкции WHERE, перечислены в табл. 3.1.

Таблица 3.1. Операции, применяемые в конструкции WHERE

Операция	Примеры использования	Назначение
=, >, <, >=, <=, <>, !=, !>, !<	<Column Name> = <Other Column Name> <Column Name> = 'Bob'	Стандартные операции сравнения. Эти операции действуют в основном так же, как и в любом другом языке программирования. Тем не менее следует подчеркнуть некоторые описанные ниже отличительные особенности. 1. Результаты выполнения операций сравнения "больше" (>), "меньше" (<) и "равно" могут изменяться в зависимости от выбранной схемы упорядочения. Например, если в базе данных выбрана схемы упорядочения, нечувствительная к регистру, то "ROMEY" = "romey", а если регистр учитывается, то "ROMEY" <> "romey". 2. И знак операции !=, и знак <> соответствуют операции сравнения на неравенство, а знаки операции !< и !> рассматриваются как соответствующие операциям сравнения "не меньше" и "не больше"
AND, OR, NOT	<Column1> = <Column2> AND <Column3> >= <Column 4> <Column1> != "MyLiteral" OR <Column2> = "MyOtherLiteral"	Стандартные булевы логические операции. Они могут использоваться для объединения нескольких условий в одной конструкции WHERE. В первую очередь выполняется операция NOT, затем AND, после чего OR. Если требуется изменить порядок выполнения операций, то можно ввести круглые скобки. Следует отметить, что операция XOR в непосредственном виде не поддерживается
BETWEEN	<Column1> BETWEEN 1 AND 5	Эта операция сравнения возвращает значение TRUE, если первое значение находится между вторым и третьим значениями включительно и является функционально эквивалентной выражению A>=B AND A<=C. В качестве любого из указанных здесь значений могут быть заданы имена столбцов, переменные или литералы
LIKE	<Column1> LIKE "ROM%"	Операция сравнения, позволяющая использовать символы % и _ в качестве подстановочных знаков. Символ % указывает, что вместо него может быть подставлено строковое значение любой длины, а символ _ указывает, что вместо него может быть подставлен любой символ. Выражение, состоящее из квадратных скобок ([]), в которые заключены символы, указывает, что для сравнения может использоваться любой отдельный символ, указанный в квадратных скобках, или любой символ из диапазона, обозначенного дефисом (-). (Например, выражение [a-c] означает, что для сравнения могут использоваться символы "a", "b" и "c", а выражение [ab] — что могут использоваться символы "a" и "b".) Символ ^, стоящий за открывающей квадратной скобкой, действует как знак операции инверсии (обращения); он указывает, что для сравнения могут использоваться все символы, кроме тех, что следуют за ним

Окончание табл. 3.1

Операция	Примеры использования	Назначение
IN	<Column1> IN (List of Numbers) <Column1> IN ("A", "b", "345")	Операция сравнения, которая возвращает TRUE, если значение, находящееся слева от ключевого слова IN, согласуется с любым из значений в списке, находящемся справа от ключевого слова IN. Как будет показано в главе 16, эта операция часто используется в подзапросах
ALL, ANY, SOME	<column|expression> (comparision operator) <ANY|SOME> (subquery)	Эти операции сравнения возвращают TRUE, если любое или все значения (в зависимости от выбранной операции) в подзапросе (subquery) соответствуют условию операции сравнения (comparision operator), например, <, >, =, >=. Ключевое слово ALL указывает, что значение должно согласовываться со всеми значениями в множестве. Операции ANY и SOME являются функциональными эквивалентами и возвращают TRUE, если значение поля в столбце (column) или выражение (expression) согласуется с каким-либо значением в множестве
EXISTS	EXISTS (subquery)	Операция сравнения, которая возвращает TRUE, если подзапрос (subquery) возвращает по крайней мере одну строку. Эта операция также рассматривается более подробно в главе 16

Конструкция ORDER BY

Результаты основного числа запросов, выполненных до сих пор, были представлены в порядке, напоминающем алфавитный, и это не случайно, что на первый взгляд может показаться неожиданным. Но дело в том, что если в запросе не указано, что результаты должны быть отсортированы в каком-то определенном порядке, получение этих данных происходит в том порядке, который выбирает сама СУБД SQL Server. Выбор применяемого по умолчанию способа сортировки всегда осуществляется на основании принятого СУБД SQL Server решения о том, какой способ сбора требуемых данных связан с наименьшими издержками. Как правило, выборка данных происходит с учетом либо физической последовательности данных в таблице, либо с учетом структуры индексов, используемых СУБД SQL Server для поиска данных.

Конструкцию ORDER BY удобно рассматривать как аналог команды "сортировать по...". Эта конструкция предоставляет возможность определить последовательность, в которой должны поступать затребованные данные. В конструкции ORDER BY можно использовать любые сочетания столбцов, при условии, что эти столбцы (указанные непосредственно или в каких-либо выражениях) находятся в таблицах, перечисленных в конструкции FROM.

Рассмотрим следующий запрос:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
```

Он должен привести к получению таких результатов:

```
Name                          ProductNumber          ReorderPoint
----------------------------  ---------------------  ------------
Adjustable Race               AR-5381                750
Bearing Ball                  BA-8327                750
...
...
Road-750 Black, 48            BK-R19B-48             75
Road-750 Black, 52            BK-R19B-52             75
(504 row(s) affected)
```

Оказалось, что результаты запроса были отсортированы в порядке возрастания значений столбца ProductID. Причина этого состоит в том, что СУБД SQL Server было принято решение по выбору такого наиболее подходящего способа выборки данных, в котором используется индекс, обеспечивающий сортировку данных в соответствии со значениями столбца ProductID. Просто так оказалось, что решение, принятое СУБД, приводит к наименьшим затратам при выполнении запроса (с точки зрения израсходованного процессорного времени и количества выполненных операций ввода-вывода). Если бы точно такой же запрос был выполнен после того, как указанная в нем таблица выросла и стала гораздо больше, то СУБД SQL Server могла бы выбрать совсем другой план выполнения и поэтому отсортировала бы данные полностью иным образом. Но мы имеем также возможность задать схемы упорядочения принудительно, изменив запрос так, чтобы он принял следующую форму:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
ORDER BY Name
```

Обратите внимание на то, что в этом запросе конструкция WHERE не потребовалась. Наличие или отсутствие этой конструкции зависит от того, с какой целью применяется запрос, но следует помнить, что если в запросе присутствует конструкция WHERE, то она должна быть задана перед конструкцией ORDER BY.

К сожалению, последний запрос в действительности не приводит к получению данных, отличных от приведенных выше, поэтому не позволяет ознакомиться с тем, как сортировка влияет на конечные результаты. Поэтому модифицируем запрос так, чтобы в нем данные сортировались по-другому — по значениям столбца ProductNumber:

```
SELECT Name, ProductNumber, ReorderPoint
FROM Production.Product
ORDER BY ProductNumber
```

Теперь результаты становятся совсем другими. В выводе присутствуют те же данные, но порядок их представления существенно изменился:

```
Name                          ProductNumber          ReorderPoint
----------------------------  ---------------------  ------------
Adjustable Race               AR-5381                750
Bearing Ball                  BA-8327                750
LL Bottom Bracket             BB-7421                375
ML Bottom Bracket             BB-8107                375
...
...
Classic Vest, L               VE-C304-L              3
Classic Vest, M               VE-C304-M              3
Classic Vest, S               VE-C304-S              3
Water Bottle - 30 oz.         WB-H098                3
(504 row(s) affected)
```

СУБД SQL Server по-прежнему выбирает метод предоставления пользователю требуемых результатов с наименьшими издержками, но конкретный набор операций, фактически осуществляемый для решения этой задачи, немного изменяется, поскольку другим становится и сам характер запроса.

Кроме того, сортировка данных может осуществляться по значениям столбцов с числовыми данными. Попытаемся применить первый из рассматриваемых запросов к базе данных `Northwind`.

В данном примере впервые используется одна из баз данных, не создаваемых по умолчанию в ходе осуществления инсталляционной процедуры. Чтобы иметь возможность выполнять следующие запросы, а также многие другие, приведенные в разных главах данной книги, необходимо установить образцы баз данных, `Northwind` и `Pubs`, с помощью дистрибутива `SQL Server 2000 Samples Install`, который можно получить с узла компании Microsoft. Инструкции по установке этих образцов приведены в приложении Д.

В ходе изучения материала в данной книге читателю придется довольно часто переходить от одной базы данных, используемой в качестве примера, к другой, а в конечном итоге даже проводить эксперименты по применению данных больше чем из одной базы данных одновременно.

При написании настоящей книги было решено предусмотреть работу с несколькими разными базами данных по многим причинам, не последней из которых было стремление ознакомить читателя с базами данных, спроектированными на основе немного разных подходов. Проблемы различий в подходах к проектированию рассматриваются более подробно в главе, посвященной проектированию.

Следующий запрос по-прежнему применяется к таблице, содержащей сведения о товарах, но имена объектов базы данных стали немного другими (обратите внимание на наличие в них буквы "s"), кроме того, указаны другая схема и другая база данных:

```
SELECT ProductID, ProductName, UnitsInStock, UnitsOnOrder
FROM Products
WHERE UnitsOnOrder > 0
AND UnitsInStock < 10
ORDER BY UnitsOnOrder DESC
```

После выполнения этого запроса будут получены такие результаты:

```
ProductID   ProductName                 UnitsInStock   UnitsOnOrder
----------  --------------------------  -------------  ------------
66          Louisiana Hot Spiced Okra   4              100
31          Gorgonzola Telino           0              70
45          Rogede sild                 5              70
21          Sir Rodney's Scones         3              40
32          Mascarpone Fabioli          9              40
74          Longlife Tofu               4              20
68          Scottish Longbreads         6              10

(7 row(s) affected)
```

В рассматриваемом запросе заслуживают внимания несколько особенностей. Во-первых, в нем используются многие конструкции, о которых шла речь в этой главе. В конструкции `WHERE` объединено несколько условий, а также предусмотрена конструкция `ORDER BY`. Во-вторых, в конструкцию `ORDER BY` введено нечто новое — ключевое слово `DESC`. Это ключевое слово сообщает СУБД SQL Server, что конструкция

ORDER BY должна действовать, сортируя данные в порядке убывания, а не в порядке возрастания значений данных, предусмотренном по умолчанию. (Чтобы явно указать, что сортировка данных должна осуществляться по возрастанию, следует ввести ключевое слово ASC.)

На этом возможности сортировки не исчерпываются. Теперь рассмотрим, как можно отсортировать данные по нескольким столбцам. Для этого достаточно ввести запятую, а за ней указать имя следующего столбца, по которому требуется выполнить сортировку.

Предположим, например, что требуется получить листинг со всеми заказами, оформленными в период с 10 по 20 декабря 1996 года. Но чтобы немного усложнить задачу, примем дополнительные предположения, что заказы необходимо отсортировать по дате, а также выполнить вторичную сортировку с учетом значений столбца CustomerID. Кроме того, просто ради интереса примем еще одно небольшое допущение: предусмотрим сортировку по значениям CustomerID в порядке убывания.

Запрос, соответствующий указанным требованиям, должен выглядеть следующим образом:

```
SELECT OrderDate, CustomerID
FROM Orders
WHERE OrderDate BETWEEN '12-10-1996' AND '12-20-1996'
ORDER BY OrderDate, CustomerID DESC
```

В данном случае полученные данные отсортированы двумя способами:

```
OrderDate                CustomerID
-----------------------  ----------
1996-12-10 00:00:00.000  FOLKO
1996-12-11 00:00:00.000  QUEDE
1996-12-12 00:00:00.000  LILAS
1996-12-12 00:00:00.000  HUNGO
1996-12-13 00:00:00.000  ERNSH
1996-12-16 00:00:00.000  BERGS
1996-12-16 00:00:00.000  AROUT
1996-12-17 00:00:00.000  SPLIR
1996-12-18 00:00:00.000  SANTG
1996-12-18 00:00:00.000  FAMIA
1996-12-19 00:00:00.000  SEVES
1996-12-20 00:00:00.000  BOTTM

(12 row(s) affected)
```

В этих результатах даты по-прежнему отсортированы в порядке возрастания (предусмотренном по умолчанию), поскольку не было дано других указаний, но анализ данных за 16-е декабря показывает, что значения CustomerID действительно отсортированы от последнего к первому, т.е. в порядке убывания.

> В рассматриваемых примерах сортировка результатов в основном осуществляется с использованием значений одного из столбцов, данные которого включаются в конечный результат, но следует учитывать, что конструкция ORDER BY может включать любой столбец любой таблицы, применяемой в запросе, независимо от того, упоминается ли имя этой таблицы в списке выборки.

Агрегирование данных с использованием конструкции GROUP BY

Описание конструкции ORDER BY в настоящей главе приведено не в той последовательности, в которой представлены конструкции в описании оператора SELECT, приведенном в начале главы. Еще раз рассмотрим полное определение структуры этого оператора:

```
SELECT <column list>
[FROM <source table(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>]
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[[FOR XML] [RAW, AUTO, EXPLICIT][, XMLDATA][, ELEMENTS][, BINARY base 64]]
[OPTION (<query hint>, [, ...n])]
```

Очевидно, что конструкция ORDER BY находится на одном из последних мест, но рассматривалась прежде, чем конструкция GROUP BY. На это есть две причины, описанные ниже.

- Конструкция ORDER BY используется гораздо чаще, чем GROUP BY, поэтому целесообразно больше времени посвятить ее изучению.
- Желательно, чтобы читатель понял, что он может произвольно соединять и согласовывать все конструкции, которые следуют за конструкцией FROM, при условии, что порядок их расположения будет соответствовать тому, который готова принять СУБД SQL Server (и который определен в описании синтаксиса).

Конструкция GROUP BY предназначена для агрегирования информации. Рассмотрим простой запрос без конструкции GROUP BY. Предположим, что требуется узнать, сколько деталей было заказано в каком-то конкретном наборе заказов:

```
SELECT OrderID, Quantity
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
```

> Обратите внимание на то, что в этом запросе применяются квадратные скобки. Как было сказано в главе 2, если в имени какого-то объекта (в данном случае таблицы) содержатся пробелы, то данное имя необходимо выделить с двух сторон с помощью квадратных скобок или одинарных кавычек; это позволяет СУБД SQL Server определить, где начинается и заканчивается имя объекта. Еще раз отметим, что автор не рекомендует использовать имена с пробелами на практике.

Выполнение этого запроса приводит к получению следующего результирующего набора:

```
OrderID     Quantity
---------   --------
11000       25
11000       30
11000       30
11001       60
```

```
11001          25
11001          25
11001          6
11002          56
11002          15
11002          24
11002          40

(11 row(s) affected)
```

Нам действительно требовалось получить только итоговые данные по трем заказам, но была выведена каждая отдельная строка расшифровки из каждого заказа. Безусловно, можно было бы получить требуемые результаты, подсчитав количество строк с помощью калькулятора, но проще воспользоваться конструкцией GROUP BY с агрегирующей функцией; в данном случае будет применяться агрегирующая функция SUM():

```
SELECT OrderID, SUM(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Выполнение этого запроса приводит к получению требуемых результатов:

```
OrderID
--------------    --------
11000             85
11001             116
11002             135

(3 row(s) affected)
```

Как и следовало ожидать, функция SUM возвращает итоговые результаты, но к чему относятся эти итоги? Если не задана конструкция GROUP BY, то результаты, полученные с помощью функции SUM, охватывают все значения из всех строк в заданном столбце. Но в данном случае конструкция GROUP BY предусмотрена, поэтому суммы, подсчитанные с помощью функции SUM, являются итогами для каждой группы.

Группирование данных с помощью конструкции GROUP BY может также осуществляться с учетом значений из нескольких столбцов. Для этого достаточно вводить запятые и задавать имена очередных столбцов.

Предположим, например, что необходимо определить номера заказов, принятых каждым служащим от заказчиков с идентификаторами CustomerID, от A до AO. Для этой цели можно воспользоваться конструкцией GROUP BY, включив в нее и столбец EmployeeID, и столбец CustomerID (назначение функции COUNT() описано ниже):

```
SELECT CustomerID, EmployeeID, COUNT(*)
FROM Orders
WHERE CustomerID BETWEEN 'A' AND 'AO'
GROUP BY CustomerID, EmployeeID
```

Выполнение этого запроса приводит к получению данных о количестве, но сами данные о количестве определяются с учетом того, сколько заказов каждый конкретный служащий принял от каждого конкретного заказчика:

```
CustomerID      EmployeeID
----------      ----------------   --
ALFKI           1                  2
ANTON           1                  1
ALFKI           3                  1
ANATR           3                  2
ANTON           3                  3
ALFKI           4                  2
ANATR           4                  1
ANTON           4                  1
ALFKI           6                  1
ANATR           7                  1
ANTON           7                  2

(11 row(s) affected)
```

Обратите внимание на то, что при использовании конструкции GROUP BY каждый столбец в списке выборки оператора SELECT должен представлять собой либо столбец, указанный в конструкции GROUP BY, либо результат агрегирования. Эти два случая рассматриваются в следующих разделах.

Результаты агрегирования

Анализ данных, которые обычно формируются с использованием конструкции GROUP BY, показывает, что эти данные представляют собой результаты агрегирования с помощью функций, воздействующих на группы данных. Например, в одном из приведенных выше запросов была получена сумма по столбцу Quantity. Эта сумма была рассчитана по выбранному столбцу и возвращена применительно к каждой группе, определенной в конструкции GROUP BY; в данном случае рассматривался только столбец OrderID. Количество предусмотренных в СУБД SQL Server агрегирующих функций весьма велико, но в данном разделе рассматриваются наиболее широко применяемые функции этого типа.

> Агрегирующие функции становятся наиболее удобными при использовании в сочетании с конструкцией GROUP BY. Но возможность их применения не ограничивается группированными запросами (запросами с конструкцией GROUP BY). Если агрегирующая функция применяется в запросе без конструкции GROUP BY, то ее действие распространяется на весь результирующий набор (на все строки, которые соответствуют конструкции WHERE). Здесь важнее всего понять, что в запросах без конструкции GROUP BY некоторые агрегирующие функции могут применяться в списке выборки только в сочетании с другими агрегирующими функциями. Иными словами, если конструкция GROUP BY не предусмотрена, то оператор выборки не может содержать агрегирующие функции, парные по отношению к именам столбцов в списке выборки. Например, если отсутствует конструкция GROUP BY, то функция AVG может применяться только в сочетании с функцией SUM, но не с определенным столбцом.

Функция AVG

Эта функция применяется для вычисления средних значений. Вначале предпримем попытку выполнить тот же запрос, что и перед этим, но на этот раз модифицируем его таким образом, чтобы получить среднее количество деталей в расчете на каждый заказ, а не общее количество по каждому заказу:

```
SELECT OrderID, AVG(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Обратите внимание на то, что полученные результаты существенно изменились:

```
OrderID
----------      ---
11000           28
11001           29
11002           33

(3 row(s) affected)
```

Эти итоговые данные можно проверить самостоятельно: в заказе номер 11000 присутствуют три разные позиции, составляющие в сумме 85, а 85 ÷ 3 = 28.33. Автор предвидит возможные возражения читателей, которые могут заявить, что полученное среднее значение равно 28.33, так зачем его округлять до 28. Эти возражения вполне оправданы, но причиной получения этих данных является то, что на них распространяются правила **приведения типа**.

Приведение типа мы рассмотрим более подробно немного позже, а пока достаточно отметить, что это — процесс, с помощью которого система автоматически преобразует один тип в другой. В данном случае расчеты в системе начинались с целых чисел, поэтому система обеспечила и возврат результатов в виде целых чисел (несмотря на то, что это привело к потере десятых долей в данных).

Функции MIN и MAX

Имена этих функций говорят сами за себя. Данные функции действительно определяют минимальное и максимальное значения для каждой группировки в выбранном столбце. Воспользуемся тем же запросом, но модифицированным с учетом применения функции MIN:

```
SELECT OrderID, MIN(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Выполнение этого запроса приводит к получению следующих результатов:

```
OrderID
----------      ---
11000           25
11001           6
11002           15

(3 row(s) affected)
```

Модифицируем этот запрос с учетом применения функции MAX:

```
SELECT OrderID, MAX(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Теперь полученные результаты выглядят так:

```
OrderID
-----------     ---
11000           30
11001           60
11002           56

(3 row(s) affected)
```

А что было бы, если бы потребовалось одновременно воспользоваться функциями MIN и MAX? Это также возможно. Достаточно включить обе эти функции в запрос:

```
SELECT OrderID, MIN(Quantity),MAX(Quantity)
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

В полученных результатах появляется дополнительный столбец, но обнаруживается определенный недостаток:

```
OrderID
-----------  -------    ----
11000        25         30
11001        6          60
11002        15         56

(3 row(s) affected)
```

Может ли читатель определить, в чем состоит этот недостаток? Было получено все, что требовалось, но теперь количество столбцов с агрегированными данными больше одного, поэтому при отсутствии контекста нелегко догадаться, для чего предназначен тот или иной столбец. Разумеется, в данном конкретном примере можно быть уверенным в том, что столбец с наибольшими значениями создан с помощью функции MAX, а столбец с наименьшими значениями — с помощью функции MIN, но не всегда можно столь же просто выяснить, к чему относится каждый столбец. Поэтому воспользуемся возможностью задать **псевдоним** (alias). Псевдонимы позволяют изменять имена столбцов в результирующем наборе и создаются с использованием ключевого слова AS, как показано ниже.

```
SELECT OrderID, MIN(Quantity) AS Minimum, MAX(Quantity) AS Maximum
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Теперь разобраться в полученных результатах становится немного проще:

```
OrderID        Minimum     Maximum
----------     --------    -------------
11000          25          30
11001          6           60
11002          15          56

(3 row(s) affected)
```

Следует отметить, что ключевое слово AS является необязательным. Но было время (предшествующее появлению версии SQL Server 6.5), когда это ключевое слово даже не рассматривалось как допустимое. Вы можете выполнить тот же запрос, что и перед этим, но удалить из текста запроса два ключевых слова AS, и вы убедитесь в том, что будет получен точно такой же результат. Следует также отметить, что псевдоним можно задать для любого столбца (и даже, как будет описано в следующей главе, для имени таблицы), а не только для результатов агрегирования.

Выполним повторно данный конкретный запрос, но на этот раз зададим псевдонимы для каждого столбца, а в некоторых позициях не будем использовать ключевое слово AS:

```
SELECT OrderID AS "Order Number", MIN(Quantity) Minimum, MAX(Quantity) Maximum
FROM [Order Details]
WHERE OrderID BETWEEN 11000 AND 11002
GROUP BY OrderID
```

Несмотря на то что в некоторых позициях ключевое слово AS пропущено, все равно имя каждого столбца изменилось в соответствии с указанными требованиями:

```
Order Number      Minimum        Maximum
------------      ----------     ------------
11000             25             30
11001             6              60
11002             15             56

(3 row(s) affected)
```

Автор должен признать, что не всегда предусматривает применение ключевого слова AS в конструкциях определения псевдонимов, но должен также отметить это как свой недостаток. Автору довелось работать с СУБД SQL Server еще задолго до того, как появилась возможность использовать ключевое слово AS, поэтому, к сожалению, он привык обходиться без этого ключевого слова (а теперь просто иногда не удается вспомнить о том, что нужно обязательно его ввести). Тем не менее читателю настоятельно рекомендуется учесть такую необходимость и использовать это с виду "лишнее" слово. Дело в том, что, во-первых, текст запроса будет выглядеть немного более понятным и, во-вторых, применение ключевого слова AS предусмотрено в стандарте ANSI.

Автор время от времени будет давать подобные рекомендации, прежде всего для того, чтобы читатель мог сразу освоить правильный способ действий (как в случае с ключевым словом AS). Но автор также хотел бы ознакомить читателя с другими вариантами применения программных конструкций, чтобы у него не возникала растерянность при встрече с такими вариантами записи операторов, которые выглядят немного иначе.

Функция COUNT(Expression|*)

Функция COUNT(*) предназначена для вычисления количества строк в результатах запроса. Для начала рассмотрим одну из наиболее широко применяемых разновидностей запроса:

```
SELECT COUNT(*)
FROM Employees
WHERE EmployeeID = 5
```

Как показано ниже, полученный при этом набор записей немного отличается от того, который был получен при использовании предыдущих запросов.

```
-----------
1

(1 row(s) affected)
```

Рассмотрим, в чем состоят эти различия. Прежде всего, как и в случае с использованием всех столбцов, возвращаемых в результате вызова функции, заданное по умолчанию имя столбца не предусмотрено, поэтому если требуется задать имя для столбца, то необходимо ввести псевдоним. Кроме того, следует отметить, что в действительности не получена достаточно значимая информация. Поэтому вначале выясним, что представляет собой этот набор записей. Он содержит количество строк, соответствующих условию WHERE запроса, относящемуся к таблице (таблицам) в конструкции FROM.

Запомните формат этого запроса. Он представляет собой основной запрос, который может служить для проверки того, что количество строк, предполагаемых для получения при выборке из таблицы и соответствующих заданным условиям WHERE, действительно является таковым.

Ради интереса попытаемся выполнить тот же запрос без конструкции WHERE:

```
SELECT COUNT(*)
FROM Employees
```

Если применительно к таблице Employees читатель не выполнял какие-либо операции удаления или вставки, то должен быть получен набор записей, который выглядит примерно таким образом:

```
-----------
9

(1 row(s) affected)
```

Что означает это число? Оно представляет собой общее количество строк в таблице Employees. Данный запрос также относится к запросам такого типа, которые следует запомнить, поскольку они потребуются в дальнейшем.

Итак, тема настоящего раздела оказалась довольно обширной! В данном разделе, посвященном описанию функции COUNT, уже были показаны два способа использования COUNT. Что касается вызова функции COUNT с параметром *, то эта тема уже описана. Теперь рассмотрим, как эта функция применяется с выражением (обычно с именем столбца).

Вначале попытаемся применить уже известный нам способ вызова на выполнение функции COUNT, но по отношению к другой таблице:

```
SELECT COUNT(*)
FROM Customers
```

Эта таблица немного больше, поэтому данные о количестве строк в ней изменяются в бо́льшую сторону:

```
-----------
91

(1 row(s) affected)
```

А теперь модифицируем этот запрос, чтобы в нем для выборки и подсчета количества строк применялся конкретный столбец:

```
SELECT COUNT(Fax)
FROM Customers
```

Полученные результаты будут немного отличаться от предыдущих:

```
-----------
69
(1 row(s) affected)
```

Предостережение. При выполнении операций агрегирования или других операций с множествами NULL-значения не учитываются.

При анализе этих новых результатов непонятно следующее: столбец Fax охватывает все строки таблицы, но данные о количестве строк в столбце Fax отличаются от количества строк в целом. Причина этого становится вполне ясной, стоит только изменить свой взгляд на вещи. Дело в том, что не в каждой строке столбца Fax значение присутствует как таковое. Иными словами, функция COUNT при ее использовании в любой форме, отличной от COUNT(*), игнорирует NULL-значения. Попытаемся проверить, действительно ли причиной расхождения являются NULL-значения:

```
SELECT COUNT(*)
FROM Customers
WHERE Fax IS NULL
```

Выполнение этого запроса приводит к получению следующего набора записей:

```
-----------
22

(1 row(s) affected)
```

Теперь рассчитаем сумму полученных значений:

```
69 + 22 = 91
```

В столбце Fax имеются 69 строк, в которых значение поля определено, и 22 строки, в которых значение в поле Fax равно NULL, что в сумме составляет 91 строку.

В действительности NULL-значения игнорируются во всех агрегирующих функциях, кроме COUNT(*). В этом вопросе следует внимательно разобраться, поскольку указанный нюанс может оказать весьма существенное влияние на полученные результаты. В частности, многие пользователи полагают, что при вычислении средних величин в столбцах с числовыми данными NULL-значения рассматриваются как равные нулю, но NULL-значения не равны нулю и не должны использоваться как таковые. Если функция AVG или другая агрегирующая функция применяется к столбцу со NULL-значениями, то эти значения не войдут в состав операции агрегирования, если с помощью каких-либо манипуляций они не будут преобразованы в значения, отличные от NULL, в пределах вызова самой функции (например, с использованием функции COALESCE() или ISNULL()). Эта тема будет рассматриваться более подробно в главе 7, но о нюансах работы со NULL-значениями всегда следует помнить, разрабатывая код T-SQL и проектируя базу данных.

С точки зрения проектирования базы данных важность этих соображений обусловлена тем, что в проекте необходимо указать, допускается ли наличие NULL-значений в каждом конкретном столбце (не считая ключевого), а от этого может зависеть способ применения к базе данных различных запросов, а также возможность получения желаемых результатов от агрегирующих функций.

Прежде чем закончить описание конструкций функции COUNT, рассмотрим, как эта функция применяется с конструкцией GROUP BY.

Предположим, что программист получил задание определить количество служащих, которые подчинены каждому руководителю. В тех операторах, которые применялись перед этим, подсчитывалось либо общее количество строк в таблице (COUNT(*)), либо количество всех строк в таблице, не имеющих NULL-значений (COUNT(ColumnName)). А после введения конструкции GROUP BY те же варианты агрегирования продолжают действовать, как и до сих пор, за исключением того, что происходит возврат данных о количестве, относящихся к каждой группировке, а не ко всей таблице. Именно этот подход можно применить для получения данных о количестве служащих, которые подчинены каждому руководителю:

```
SELECT ReportsTo, COUNT(*)
FROM Employees
GROUP BY ReportsTo
```

Обратите внимание на то, что группирование осуществляется только по столбцу ReportsTo, а функция COUNT() выполняет роль агрегирующей функции, поэтому включать используемый в ней столбец в конструкцию GROUP BY не требуется.

```
ReportsTo
-------------    --
NULL             1
2                5
5                3

(3 row(s) affected)
```

На основании полученных результатов можно судить о том, что руководитель с идентификатором ManagerID, равным 2, имеет в подчинении пять служащих, а руководителю с идентификатором ManagerID, равным 5, подчиняются трое служащих. Кроме того, можно сделать вывод, что в одной записи Employees в поле ReportsTo содержится NULL-значение; на этом основании можно сделать вывод, что данный служащий никому не подчиняется (по-видимому, это — президент компании).

Очевидно, следует отметить, что с формальной точки зрения конструкция GROUP BY может использоваться без каких-либо агрегирующих функций, но такой способ ее применения не имеет смысла. Дело в том, что в этом случае СУБД SQL Server обрабатывает последовательно все строки в целях их группирования, но с точки зрения осуществляемых при этом действий тот же результат может быть получен с применением опции DISTINCT (которая вскоре будет описана), а быстродействие при этом становится намного выше.

Теперь, после описания работы с группами, перейдем к рассмотрению одного из понятий, при освоении которых у большинства программистов возникают сложности. Надеемся, что после чтения следующего раздела у читателя такое впечатление полностью рассеется.

Распространение условий на группы с помощью конструкции HAVING

До сих пор все рассматриваемые условия распространялись на конкретные строки. Таким образом, если в каком-то поле строки не содержалось определенное значение или если это значение не находилось в указанных пределах, то вся строка рассматривалась как не соответствующая заданным условиям. Проверка условий такого рода происходит без учета какого-либо группирования строк.

Тем не менее иногда возникает необходимость сгруппировать с учетом определенных условий целый ряд строк. Иными словами, требуется ввести в состав определенной группы все относящиеся к ней строки, но применить заданное условие только после полного накопления всех групп. Именно для этой цели предназначена конструкция HAVING.

Конструкция HAVING используется, только если в запросе имеется также конструкция GROUP BY. Еще раз отметим, что конструкция WHERE применяется к каждой строке еще до того, как эта строка получит шанс войти в состав группы, а конструкция HAVING — к агрегированному значению для этой группы.

Начнем с внесения небольшого изменения в запрос GROUP BY, который рассматривался в конце предыдущего раздела; в этом запросе требовалось узнать, какое количество служащих подчиняется каждому руководителю, указанному с помощью идентификатора ManagerID.

```
SELECT ReportsTo AS Manager, COUNT(*) AS Reports
FROM Employees
GROUP BY ReportsTo
```

Теперь еще раз рассмотрим полученные результаты:

```
Manager     Reports
---------   -------
NULL        1
2           5
5           3

(3 row(s) affected)
```

В следующей главе будет показано, как заменить идентификаторы EmployeeID руководителей, которые находятся в столбце Manager, именами этих руководителей. Но на данный момент отметим, что, согласно полученным результатам, в данной компании имеются только два руководителя. Очевидно, что каждый из служащих подчиняется этим двум сотрудникам. Исключение составляет только один человек, которому не назначен руководитель; по-видимому, он является президентом компании (можно было бы написать запрос для проверки этого предположения, но пока просто будем считать, что это предположение — правильное).

В данный запрос не включена конструкция WHERE, поэтому конструкция GROUP BY применяется к каждой строке таблицы и в группировании участвуют все строки. Для того чтобы проверить, какое влияние это окажет на итоговые данные, введем в запрос конструкцию WHERE:

```
SELECT ReportsTo AS Manager, COUNT(*) AS Reports
FROM Employees
WHERE EmployeeID != 5
GROUP BY ReportsTo
```

В результате возникает одно небольшое изменение, которого и следовало ожидать:

```
Manager     Reports
----------  ---------
NULL        1
2           4
5           3

(3 row(s) affected)
```

Распределение по группам почти не изменилось, но одна строка была исключена еще до того, как в запросе началась обработка конструкции GROUP BY. Очевидно, что с помощью конструкции WHERE была исключена одна строка со значением EmployeeID, равным 5. Оказалось, что сотрудник с идентификатором EmployeeID, равным 5, подчиняется сотруднику с идентификатором ManagerID, равным 2. После того как строка со значением 5 в поле EmployeeID перестала быть частью запроса, количество строк, относящихся к группе, со значением ManagerID, равным 2, уменьшилось на единицу.

Теперь рассмотрим выполняемые действия с другой точки зрения. Попытайтесь ответить на следующий вопрос: "Какие руководители имеют в своем подчинении больше четырех человек?" Безусловно, можно выполнить запрос без конструкции WHERE и проверить соответствие этому условию данных в каждой строке результата, но как это сделать программным путем? Иными словами, как составить запрос, который возвращает данные только о тех руководителях, которым подчиняются больше четырех служащих? При попытке решить эту задачу с помощью конструкции WHERE обнаруживается, что такой подход не позволяет определить количество строк в группе, к которой применяется какая-либо функция агрегирования, поскольку выполнение конструкции WHERE завершается системой еще до того, как начинается агрегирование. Именно поэтому предусмотрена конструкция HAVING:

```
SELECT ReportsTo AS Manager, COUNT(*) AS Reports
FROM Employees
GROUP BY ReportsTo
HAVING COUNT(*) > 4
```

После выполнения этого запроса обнаруживается, что полученные результаты действительно позволяют решить указанную задачу:

```
Manager     Reports
--------    -------
2           5

(1 row(s) affected)
```

Очевидно, что в компании имеется только один руководитель, которому подчиняются больше четырех служащих.

Как уже было сказано, ту же информацию можно было бы получить, просмотрев ранее полученные результаты. Но подобные итоговые данные не всегда являются столь краткими. К тому же, применяя указанный способ наложения условия на сгруппированные строки с помощью конструкции HAVING, часто можно быстро получить точный ответ, не требующий дальнейшего анализа.

Теперь рассмотрим более крупный набор записей, а затем оставим эту тему до следующей главы, где речь пойдет о многотабличных запросах. Допустим, что требуется

составить запрос, который определял бы общее количество заказанных товаров по всем заказам, хранящимся в системе. Очевидно, что это довольно несложный запрос:

```
SELECT OrderID, SUM(Quantity) AS Total
FROM [Order Details]
GROUP BY OrderID

OrderID        Total
-----------    -------
10248          27
...
...
11075          42
11076          50
11077          72

(830 row(s) affected)
```

Но, к сожалению, при этом формируется очень длинный список, анализ которого является весьма затруднительным. Поэтому воспользуемся возможностями СУБД SQL Server, чтобы немного сократить этот список и упростить задачу его анализа. Предположим, что нас интересуют только заказы с большим количеством товаров. Скажем, требуется создать запрос, который возвращал бы ту же информацию, но ограничивался теми заказами, в которых общее количество заказанных единиц товаров превышало 300. Это можно легко сделать, введя дополнительно конструкцию HAVING:

```
SELECT OrderID, SUM(Quantity) AS Total
FROM [Order Details]
GROUP BY OrderID
HAVING SUM(Quantity) > 300
```

После этого список становится намного короче:

```
OrderID        Total
---------      -------
10895          346
11030          330

(2 row(s) affected)
```

Вполне очевидно, что, налагая различные условия на формируемые группы, можно сокращать этот список для получения лишь интересующих нас строк. Получив такую промежуточную информацию, можно перейти к изучению именно тех заказов, которые имеют идентификаторы OrderID, равные 10895 и 11030, или, как будет описано в следующих главах, применить операцию JOIN, чтобы соединить данные, полученные из одной таблицы, с данными другой таблицы, и получить еще более детализированную информацию.

Вывод кода XML с использованием конструкции FOR XML

Ко времени выхода предыдущей версии СУБД SQL Server в 2000 году язык XML еще не имел широкого распространения, но уже показал себя как один из основных способов обеспечения доступа к данным. Поэтому компания Microsoft, готовясь к выпуску указанной версии СУБД SQL Server (SQL Server 2000), предусмотрела воз-

можность вывода результатов в формате XML, а не только оформления их в виде традиционного результирующего набора. Указанные средства вывода данных показали себя как чрезвычайно мощные, особенно в такой среде, как Web, или в системах, основанных на использовании нескольких разных платформ.

С тех пор специалисты компании Microsoft немного усовершенствовали способы вывода данных в формате XML, но основы этих способов остались теми же, а их значимость еще больше возросла. Описание всего того, что касается языка XML, представляет собой отдельную большую тему, поэтому в настоящей главе не будут даны подробные сведения об использовании конструкции FOR XML, но в главе 16 средства XML будут рассмотрены гораздо подробнее. А пока просто поверьте автору, что сначала лучше изучить основы.

Использование подсказок, сформированных с помощью конструкции OPTION

Конструкция OPTION применяется для передачи СУБД SQL Server указаний, касающихся выбора способа выполнения запроса. Но в действительности СУБД SQL Server почти всегда находит гораздо лучший способ выполнения запроса, чем программист, поэтому чаще всего в результате использования конструкции OPTION производительность системы снижается. Однако иногда применение конструкции OPTION действительно необходимо.

Это – еще одна из тем, которым будет уделено больше внимания позднее. Подсказки, применяемые в запросах, будут рассматриваться более подробно в главах, в которых речь пойдет о блокировках, а поскольку мы еще не описали, как влияют подсказки на выполнение запросов, то не сформированы предпосылки для понимания конструкции OPTION, поэтому пока отложим также обсуждение и этой темы.

Предикаты DISTINCT и ALL

В данном разделе мы рассмотрим последнюю важную тему, которая относится к оператору SELECT, после чего перейдем к рассмотрению операторов действия (которые предназначены для внесения изменений в данные). Предикаты DISTINCT и ALL позволяют выполнять операции над повторяющимися данными.

Предположим, например, что требуется составить список идентификаторов поставщиков всех товаров, имеющихся на складе. Эту информацию можно легко получить из таблицы Products с помощью следующего запроса:

```
SELECT SupplierID
FROM Products
WHERE UnitsInStock > 0
```

При этом будут получены результаты, в которых каждая отдельная строка соответствует одному идентификатору SupplierID для каждой строки в таблице Products:

```
SupplierID
--------------
1
1
1
2
3
```

```
3
3
...
...
12
23
12

(72 row(s) affected)
```

Разумеется, формально стоящая перед нами задача выполнена, но в действительности требовалось получить нечто иное. Достаточно лишь взглянуть на то, сколько строк здесь содержат одни и те же данные! По результатам выполнения других запросов, которые рассматривались в этой главе, можно убедиться в том, что данная конкретная таблица невелика, однако количество возвращенных строк и дубликатов буквально ошеломляет. Но и в данном случае, как и при решении других проблем, с которыми мы сталкивались в данной главе, существует эффективное решение. Оно имеет форму предиката DISTINCT, который может применяться в операторе SELECT.

Попытаемся повторно выполнить приведенный выше запрос, внеся в него небольшое изменение:

```
SELECT DISTINCT SupplierID
FROM Products
WHERE UnitsInStock > 0
```

Теперь сформирован действительно удобный список идентификаторов SupplierID с данными о тех поставщиках, товары которых имеются на складе:

```
SupplierID
--------------
1
2
3
...
...
27
28
29

(29 row(s) affected)
```

Вполне очевидно, что благодаря применению предиката DISTINCT размеры списка существенно сократились, а само содержимое списка стало гораздо более удобным в использовании. Еще одним преимуществом данного запроса является то, что он фактически обладает более высокой производительностью, чем первоначальный запрос. Проблемы повышения производительности будут рассматриваться в следующих главах, но на данный момент достаточно сказать, что повышение производительности при использовании предиката DISTINCT обусловлено тем, что СУБД SQL Server не приходится возвращать каждую отдельную строку, поэтому уменьшается объем работы, требуемый для выполнения запроса в указанной форме.

Но на этом возможности предиката DISTINCT далеко не исчерпываются, и можно убедиться в том, что многие программисты, работающие на языке SQL, об этом даже не подозревают. Дело в том, что ключевое слово DISTINCT может не только применяться

в качестве предиката в операторе SELECT, но и входить в выражение, которое служит параметром для агрегирующей функции. Рассмотрим три приведенных ниже запроса.

Вначале определим количество строк в таблице Order Details базы данных Northwind:

```
SELECT COUNT(*)
FROM [Order Details]
```

Если в таблицу Order Details не вносили изменения, то выполнение этого запроса должно показать, что в данной таблице имеются 2155 строк.

Теперь выполним тот же запрос, используя для подсчета количества строк конкретный столбец:

```
SELECT COUNT(OrderID)
FROM [Order Details]
```

Столбец OrderID входит в состав ключа для данной таблицы, поэтому не может содержать какие-либо NULL-значения (дополнительные сведения по этой теме приведены в главе, посвященной индексации). Поэтому общее количество строк, полученное с помощью данного запроса, всегда должно совпадать с тем количеством, которое обнаруживается с помощью выражения COUNT(*); и в данном случае оно равно 2155.

Ключ — это столбец (или комбинация столбцов), который может использоваться для идентификации любой строки в таблице. В действительности ключи подразделяются на несколько типов (дополнительная информация об этом приведена в главах 7–9), но обычно, если термин "ключ" применяется без уточнений, под ним подразумевается первичный ключ таблицы. *Первичным ключом* называется столбец (или комбинация столбцов), который содержит данные, однозначно идентифицирующие каждую строку. Это означает, что, ссылаясь на некоторую строку с использованием относящегося к ней первичного ключа, можно быть уверенным в получении только одной строки, поскольку не допускается, чтобы в одной и той же таблице две строки имели одинаковый первичный ключ.

Теперь рассмотрим тот способ применения ключевого слова DISTINCT, о котором шла речь выше. Еще раз модифицируем рассматриваемый запрос:

```
SELECT COUNT(DISTINCT OrderID)
FROM [Order Details]
```

После этого будет получен совсем другой результат:

```
-----------
830

(1 row(s) affected)
```

Все строки с повторяющимися значениями в поле OrderID исключены еще до выполнения операции агрегирования, поэтому количество строк стало намного меньше.

Следует отметить, что ключевое слово DISTINCT может использоваться с любой функцией агрегирования, но можно поставить под сомнение то, будут ли при этом иметь многие из этих функций какое-либо практическое значение. Например, трудно представить себе, для чего может потребоваться вычисление средних значений только по строкам, оставшимся после применения предиката DISTINCT.

Теперь перейдем к рассмотрению предиката ALL. На практике чрезвычайно редко приходится сталкиваться с тем, что в каком-то операторе применяется предикат ALL, за одним исключением. По-видимому, проще всего понять назначение предиката ALL, рассматривая его как противоположный предикату DISTINCT. Как уже было сказано, предикат DISTINCT обеспечивает исключение строк с повторяющимися данными, а предикат ALL используется в качестве указания, что должна быть включена каждая строка. Предикат ALL применяется по умолчанию во всех операторах SELECT, за исключением тех ситуаций, когда выполняется операция UNION. Последствия применения предиката ALL в сочетании с операцией UNION рассматриваются в следующей главе, но на данный момент достаточно понять, что выборка данных по такому принципу, который предусмотрен в предикате ALL, происходит во всех тех случаях, когда не используется предикат DISTINCT.

Внесение данных с помощью оператора INSERT

В предыдущих разделах приведен довольно большой объем информации об операторах SELECT. Тем не менее применение этих операторов не имеет смысла, если в базу данных не введены перед этим какие-либо данные. Именно для этой цели служит оператор INSERT.

Оператор INSERT имеет примерно такой основной синтаксис:

```
INSERT [INTO] <table> [(column_list)] VALUES (data_values)
```

Рассмотрим последовательно структуру этого оператора.

Прежде всего отметим, что INSERT — оператор действия. Само ключевое слово INSERT сообщает СУБД SQL Server, что должна быть выполнена вставка данных, а все, что следует за этим ключевым словом, является уточнением деталей требуемого действия.

Ключевое слово INTO главным образом предназначено лишь для уточнения. Его единственным назначением является повышение удобства чтения оператора. Без ключевого слова INTO можно полностью обойтись, но автор настоятельно рекомендует использовать его именно по той причине, по которой оно было предусмотрено в определении оператора, — благодаря ему значительно повышается удобство чтения. Рекомендуем читателю в ходе изучения данного раздела попытаться вызвать на выполнение некоторые операторы и с ключевым словом INTO, и без него. Безусловно, отказ от применения INTO приводит к небольшому сокращению объема кода, но вместе с тем чтение кода в значительной степени затрудняется. Тем не менее разработчик вправе сам принимать решение о его использовании.

За указанными ключевыми словами следует имя таблицы, в которую должны быть вставлены данные.

До того как мы подошли к изучению данной части оператора INSERT, все обстояло довольно просто, а теперь нам придется столкнуться с более сложной его частью — с определением списка столбцов. Явно заданный список столбцов (в котором должен быть конкретно указан каждый столбец, принимающий вводимые значения) является необязательным, но если это явное определение не используется, то приходится соблюдать исключительную осторожность. Если явно заданный список столбцов не предусмотрен, то предполагается, что каждое значение в операторе INSERT должно

соответствовать столбцу, находящемуся в той же порядковой позиции в строке таблицы, что и само вводимое значение (первое значение вводится в первый столбец, второе значение — во второй и т.д.). Кроме того, значение должно быть задано для каждого столбца, который не принимает NULL-значений и не имеет значения, заданного по умолчанию (пояснения к сказанному будут даны немного позже), пока не будет достигнут последний столбец. Короче говоря, в этой части оператора должен быть приведен список из одного или нескольких столбцов, данные для заполнения которых будут приведены в следующей части оператора.

Наконец, задаются значения, которые должны быть вставлены в базу данных. Для этой цели могут применяться два способа, но пока мы остановимся на рассмотрении такого способа, в котором осуществляется вставка одной строки с использованием явно приведенных данных. Для того чтобы установить значения, необходимо прежде всего ввести ключевое слово `VALUES`, а затем представить список значений, разделенных запятыми, который заключен в круглые скобки. Количество элементов в списке значений должно точно совпадать с количеством столбцов в списке столбцов. Тип данных каждого значения также должен совпадать или допускать неявное преобразование в тип данных столбца, которому соответствует это значение (сопоставление столбцов и вставляемых в них значений осуществляется с учетом порядка следования).

Иногда разработчики задумываются над тем, следует ли задавать значение конкретно для каждого столбца. Но фактически рекомендуется при любых обстоятельствах указывать каждый столбец, применяемый для вставки данных, даже если используемое по умолчанию значение вставляется с помощью ключевого слова `DEFAULT` или явно задается NULL-значение. Ключевое слово `DEFAULT` сообщает СУБД SQL Server, что должно быть вставлено то значение, которое предусмотрено по умолчанию для данного столбца (если такое значение не предусмотрено, появляется ошибка).

Соблюдение указанной рекомендации способствует повышению удобства кода для чтения, поскольку позволяет полностью определить, какие данные должны быть введены в таблицу. Кроме того, по мнению автора, если каждый столбец перечисляется явно, то количество возможных программных ошибок уменьшается.

Безусловно, приведенное выше описание может показаться сложным, поэтому рассмотрим некоторые практические примеры. Начнем с использования базы данных `pubs`. Это — еще одна база данных, которая широко используется в данной книге. Развертывание этой базы в системе происходит при инсталляции предлагаемых компанией Microsoft образцов баз данных, в число которых входит `Northwind`. Для перехода к этой базе данных не забудьте внести изменение в окне Query или выполните команду `USE Pubs`.

Основная часть операций вставки, которые будут описаны в этой главе, относится к таблице `stores` (это — очень аккуратная и простая таблица, с которой удобно начать), поэтому рассмотрим свойства этой таблицы (рис. 3.3). Для этого в окне Object Explorer программы Management Studio разверните узел Tables, относящийся к базе данных `pubs`. После этого разверните также узел Columns (см. рис. 3.3).

Очевидно, что в этой таблице каждый столбец имеет тип `char` или `varchar`.

В первом примере применения операции вставки исключим необязательный список столбцов и предоставим СУБД SQL Server возможность действовать исходя из предположения, что какое-то значение предусмотрено для каждого столбца:

```
INSERT INTO stores
VALUES ('TEST', 'Test Store', '1234 Anywhere Street', 'Here', 'NY', '00319')
```

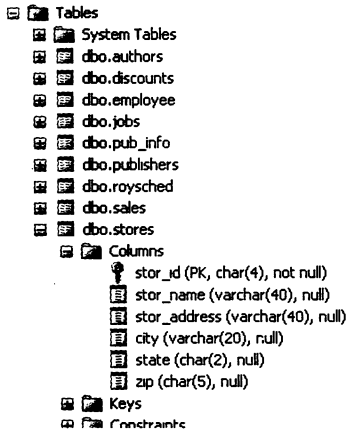


Рис. 3.3. Список столбцов таблицы **stores**

Как было сказано выше, если отдельный список столбцов не предусмотрен (вскоре будет описано, как предусмотреть в операторе список столбцов), то все значения должны быть заданы в том же порядке, в каком определены соответствующие им столбцы в таблице. После выполнения этого запроса должно появиться сообщение о том, что данный запрос оказал воздействие на одну строку. А теперь, ради интереса, попытаемся выполнить точно такой же запрос во второй раз. Будет получено следующее сообщение об ошибке:

```
Msg 2627, Level 14, State 1, Line 1
Violation of PRIMARY KEY constraint 'UPK_storeid'. Cannot insert duplicate
key in object 'dbo.stores'.
The statement has been terminated.
```

Очевидно, что запрос был выполнен успешно в первый раз, но не во второй. Причина этого состоит в том, что на данной таблице задан первичный ключ, который не допускает появления дубликатов значений в столбце `stor_id`. А если бы было изменено значение только в одном поле, соответствующем этому столбцу, то можно было бы оставить неизменными значения всех других полей, и вставка новой строки была бы выполнена успешно. Дополнительные сведения о первичных ключах приведены в тех главах, в которых рассматриваются вопросы проектирования и ограничения целостности.

С помощью следующего запроса рассмотрим, к чему привело выполнение оператора вставки:

```
SELECT *
FROM stores
WHERE stor_id = 'TEST'
```

Выполнение этого запроса приводит к получению именно тех данных, которые были вставлены в таблицу:

```
stor_id     stor_name     stor_address           city     state    zip
-------     ----------    --------------------   -----    -----    ----
TEST        Test Store    1234 Anywhere Street   Here     NY       00319

(1 row(s) affected)
```

Следует отметить, что в конце каждого столбца в этом листинге было удалено несколько пробелов, чтобы результаты аккуратно разместились на странице, но это не повлияло на представление самих данных, которые полностью соответствуют нашим ожиданиям.

Теперь еще раз попытаемся выполнить тот же запрос после внесения в него изменений, касающихся отдельных столбцов:

```
INSERT INTO stores
  (stor_id, stor_name, city, state, zip)
VALUES
  ('TST2', 'Test Store', 'Here', 'NY', '00319')
```

Обратите внимание на то, что в строке со значениями данных изменены только два элемента. Во-первых, модифицировано значение, вставляемое в столбец первичного ключа, чтобы не вырабатывалась ошибка. Во-вторых, оставлено пустым значение, которое должно быть связано со столбцом `stor_address`, поскольку имя этого столбца исключено из списка столбцов. Исключать столбец из списка столбцов и не предоставлять для него никаких данных в операторе `INSERT` допускается в нескольких ситуациях. Но в данный момент мы лишь пользуемся тем фактом, что столбец `stor_address` не является обязательным, иными словами, он допускает наличие NULL-значений. В данном случае значение для столбца не предоставляется, а заданное по умолчанию значение для него не предусмотрено (дополнительные сведения о значениях, предусмотренных по умолчанию, будут приведены ниже), поэтому соответствующему полю столбца `stor_address` при выполнении данного оператора вставки будет присвоено NULL-значение. Проверим, так ли это, повторно выполнив тот же проверочный оператор `SELECT` с одним небольшим изменением, касающимся значения первичного ключа:

```
SELECT *
FROM stores
WHERE stor_id = 'TST2'
```

Теперь полученные данные выглядят немного иначе:

```
stor_id    stor_name    stor_address    city    state    zip
---------  -----------  --------------  ------  -------  ------
TST2       Test Store   NULL            Here    NY       00319

(1 row(s) affected)
```

Обратите внимание на то, что вместо пропущенного значения поля вставлено `NULL`.

Следует отметить, что описанные выше действия можно выполнять только по отношению к столбцам, **принимающим неопределенные значения**. Такая характеристика столбца в основном не требует пояснений и просто означает, что в данном столбце допускается наличие NULL-значений. Вопросы использования неопределенных значений (NULL-значений) будут подробно рассмотрены в следующих главах, поэтому сейчас лишь отметим, что одни столбцы допускают наличия в них NULL-значений, а другие — нет. А что касается столбцов, допускающих наличие NULL-значений, то всегда возможно не предоставлять для них информацию, вставляя данные в другие столбцы.

Но если столбец не принимает неопределенное значение, то должно соблюдаться одно из трех перечисленных ниже условий, поскольку в противном случае будет получено сообщение об ошибке, а оператор INSERT отвергнут.

- Столбец определен со **значением, заданным по умолчанию**. Значение, заданное по умолчанию, представляет собой постоянное значение, вставка которого выполняется, если не предусмотрено какое-либо другое значение. Информация о том, как определять значения, заданные по умолчанию, приведена в главе 7.
- Столбец определен как предназначенный для получения значения, вырабатываемого системой в той или иной форме. Наиболее часто таковым является значение IDENTITY (рассматриваемое более подробно в главе, посвященной проектированию). Заполняя этот столбец, система, как правило, начинает отсчет в первой строке с 1, увеличивает это значение до 2, заполняя вторую строку, и т.д. Значения IDENTITY фактически нельзя рассматривать как "номера строк", поскольку в дальнейшем строки могут быть удалены, а при определенных обстоятельствах происходит пропуск некоторых из этих значений, т.е. строго последовательная нумерация нарушается, но они оправдывают свое назначение в том, что обеспечивают наличие в каждой строке ее собственного уникального идентификатора.
- Для столбца задается значение.

Исключительно ради полноты описания рассмотрим еще один оператор INSERT. Но на этот раз речь идет о вставке информации о новой торговой сделке в таблицу sales. Для просмотра свойств таблицы sales можно либо открыть относящееся к ней окно Properties, как в описанном выше случае с таблицей stores, либо вызвать на выполнение системную хранимую процедуру sp_help. С помощью процедуры sp_help может быть получена информация о любом объекте базы данных, типе данных, определяемом пользователем, или о типе данных SQL Server. Синтаксис оператора вызова процедуры sp_help является следующим:

```
EXEC sp_help <name>
```

Поэтому для просмотра свойств таблицы sales достаточно ввести следующую команду в программе Query Analyzer:

```
EXEC sp_help sales
```

Выполнение этой команды приведет к получению, в частности, таких результатов:

```
Column_name     Type          Length        Nullable
-------------   ---------     --------      --------
stor_id         char          4             no
ord_num         varchar       20            no
ord_date        datetime      8             no
qty             smallint      2             no
payterms        varchar       12            no
title_id        tid           6             no
```

Таблица sales включает шесть столбцов, но особого внимания читателя заслуживают столбцы qty и ord_date, поскольку они имеют типы, которые нам еще не приходилось применять в операторах вставки до этого момента (столбец title_id относится к типу tid, но фактически это просто тип, определяемый пользователем, который представляет собой символьный тип длиной 6).

Отличительной особенностью данного запроса является то, что он показывает, в каком формате следует представлять данные, вводимые в базу данных с учетом их типа. В частности, при вводе числовых значений не должны использоваться кавычки, как при вводе символьных данных. Но для данных типа `datetime` кавычки требуются (на самом деле, эти данные вводятся в виде строковых, а затем преобразуются в данные типа `datetime`).

```
INSERT INTO sales
   (stor_id, ord_num, ord_date, qty, payterms, title_id)
VALUES
   ('TEST', 'TESTORDER', '01/01/1999', 10, 'NET 30', 'BU1032')
```

После ввода этого запроса появляется уже знакомое нам сообщение "`(1 row(s) affected)`".

> Следует отметить, что в данном примере использовался формат `MM/DD/YYYY`, который широко распространен в США, но с одинаковым успехом можно применять и многие другие форматы (например, более часто встречающийся во многих странах мира формат `YYYY-MM-DD`). На сервере используется заданное по умолчанию значение формата, зависящее от того, была ли сразу приобретена локализованная копия СУБД SQL Server или значение этого параметра было изменено на сервере.

Оператор INSERT INTO. . . SELECT

Безусловно, режим работы, при котором происходит единовременно вставка только одной строки, вполне приемлем и удобен, но иногда возникает необходимость во вставке целого блока данных. В настоящей книге будет продемонстрировано несколько сценариев, в которых вставка может осуществляться таким образом, но на данный момент сосредоточимся на таком варианте, что данные, вставляемые в таблицу, формируются из других источников, подобных перечисленным ниже.

- Другая таблица в той же базе данных.
- Полностью иная база данных на том же сервере.
- Разнородный запрос на выборку информации из другой СУБД SQL Server или другие данные.
- Та же таблица (в таком случае в операторе `SELECT` обычно предусмотрено выполнение над данными каких-либо математических операций или внесение в данные других изменений).

Все эти действия позволяет выполнять оператор `INSERT INTO ... SELECT`. По своему синтаксису этот оператор представляет собой комбинацию двух операторов, описанных выше в данной главе, — `INSERT` и `SELECT`. Данный синтаксис выглядит примерно так:

```
INSERT INTO <table name>
[<column list>]
<SELECT statement>
```

В качестве данных, вводимых в операторе `INSERT`, используется результирующий набор, созданный в результате выполнения оператора `SELECT`.

Проверим работу оператора INSERT INTO...SELECT на примере, который очень часто встречается в практике программирования, — выборки данных, хранящихся во временной таблице. В этом случае будет объявлена переменная типа таблицы и заполнена строками данных из таблицы Orders, как показано ниже.

> Рассматриваемый в данном примере блок кода называется *сценарием*. Этот конкретный сценарий состоит из одного пакета. Более подробные сведения о пакетах приведены в главе 11.

```
/* Следующий оператор задает Northwind в качестве текущей базы данных.
** Это позволяет обеспечить переход к использованию нужной базы данных
** непосредственно в коде сценария
*/
USE Northwind
/* В следующем операторе содержится объявление рабочей таблицы.
** Данная конкретная таблица представляет собой переменную типа таблицы,
** которая объявляется динамически
*/
DECLARE @MyTable Table
(
  OrderID     int,
  CustomerID  char(5)
)
/* После объявления переменной типа таблицы можно приступить к заполнению ее
** данными с помощью оператора SELECT. Следует учитывать, что в данном
** случае можно было бы с таким же успехом вставлять данные в постоянную
** таблицу (а не в переменную типа таблицы)
*/
INSERT INTO @MyTable
  SELECT OrderID, CustomerID
  FROM Northwind.dbo.Orders
  WHERE OrderID BETWEEN 10240 AND 10250
-- Наконец, проверим, удалось ли достичь требуемых результатов вставки данных
SELECT *
FROM @MyTable
```

После выполнения этого кода будут получены следующие результаты:

```
(3 row(s) affected)
OrderID        CustomerID
-------        ----------
10248          VINET
10249          TOMSP
10250          HANAR

(3 row(s) affected)
```

В данном случае обнаруживаются два сообщения "(3 row(s) affected)". Первое из этих сообщений является результатом промежуточного этапа выполнения оператора INSERT...SELECT — выборки оператором SELECT трех строк, которые должны быть вставлены в таблицу. Для проверки того, какие данные вставлены в таблицу, в этом сценарии затем непосредственно используется оператор SELECT.

Следует отметить, что при попытке применить оператор SELECT к отдельно взятой таблице @MyTable (т.е. рассматриваемой за пределами данного сценария) возникает ошибка. Дело в том, что @MyTable — это переменная, объявленная в сценарии, которая существует только в течение прогона пакета. После этого она автоматически уничтожается.

Заслуживает также внимания то, что в данном сценарии можно было бы использовать так называемую временную таблицу. Вообще говоря, такие таблицы по своему характеру аналогичны переменным, но действуют не совсем так же. Дополнительная информация о временных таблицах и переменных типа таблиц приведена в главах 11–13.

Модификация данных с помощью оператора UPDATE

Оператор UPDATE, как и большинство операторов SQL, в основном выполняет такие действия, о которых говорит само его имя, — обновляет существующие данные. Структура этого оператора имеет небольшие отличия по сравнению с оператором SELECT, но между этими двумя операторами можно обнаружить определенные аналогии. Синтаксис оператора UPDATE выглядит так:

```
UPDATE <table name>
SET <column> = <value> [,<column> = <value>]
[FROM <source table(s)>]
[WHERE <restrictive condition>]
```

Данные, применяемые в операторе UPDATE, могут быть получены из нескольких таблиц, но затем они применяются только к одной таблице. Иными словами, в этом операторе могут быть предусмотрены условия, в которых упоминаются несколько разных таблиц, или может осуществляться выборка значений из многих разных таблиц, но объектом действия по обновлению одновременно может служить только одна таблица. На данном этапе организация работы по указанному принципу не будет рассматриваться слишком подробно, поскольку тема, касающаяся соединения нескольких таблиц, еще не рассматривалась (речь об этом пойдет в следующей главе), поэтому в данном разделе не приведены сложные операторы UPDATE. Рассмотрим несколько простых примеров обновления.

Начнем с применения операторов обновления к данным, которые были вставлены в таблицы после выполнения примеров, рассматривавшихся в разделе с описанием оператора INSERT. Выполним повторно один из встречавшихся ранее запросов, чтобы определить, как выглядит одна строка вставленных данных (не забудьте снова переключиться на базу данных pubs):

```
SELECT *
FROM stores
WHERE stor_id = 'TEST'
```

После выполнения этого запроса будут получены следующие результаты:

```
stor_id    stor_name     stor_address          city     state    zip
---------  -----------   --------------------  ------   ------   -----
TEST       Test Store    1234 Anywhere Street  Here     NY       00319
```

Обновим значение в столбце city:

```
UPDATE stores
SET city = 'There'
WHERE stor_id = 'TEST'
```

Как и в случае с использованием оператора INSERT, ответ, полученный от СУБД SQL Server, довольно лаконичен:

```
(1 row(s) affected)
```

Но после повторного выполнения того же оператора SELECT обнаруживается, что заданное значение действительно изменилось:

```
stor_id     stor_name    stor_address          city    state   zip
----------  -----------  --------------------  -----   ------  ----
TEST        Test Store   1234 Anywhere Street  There   NY      00319
```

Следует отметить, что в одном операторе можно внести изменения сразу в несколько полей. Для этого достаточно добавить запятую и предусмотреть дополнительное выражение с указанием столбца. Например, следующий оператор позволяет обновить значения в двух столбцах:

```
UPDATE stores
SET city = 'There', state = 'CA'
WHERE stor_id = 'TEST'
```

При желании в конструкции SET вместо явно заданных значений, которые использовались до сих пор, можно привести выражение. В качестве примера рассмотрим несколько записей из таблицы titles базы данных pubs:

```
SELECT title_id, price
FROM titles
WHERE title_id LIKE 'BU%'
```

В данном случае операция LIKE используется для выборки строк, в которых значения в столбце title_id начинаются с подстроки BU, за которой могут следовать любые символьные данные (на что указывает подстановочный символ %). При условии, что с данными в базе данных pubs не проводились какие-либо манипуляции, должны быть получены примерно такие результаты:

```
title_id    price
----------  -----------
BU1032      19.9900
BU1111      11.9500
BU2075      2.9900
BU7832      19.9900

(4 row(s) affected)
```

Ознакомившись с тем, как выглядят исходные данные, предпримем попытку применить другой способ обновления, в котором в операторе UPDATE используется выражение:

```
UPDATE titles
SET price = price * 1.1
WHERE title_id LIKE 'BU%'
```

После выполнения этого оператора обновления снова введем тот же оператор SELECT:

```
SELECT title_id, price
FROM titles
WHERE title_id LIKE 'BU%'
```

Должно быть обнаружено, что для каждого идентификатора title_id, который начинается с подстроки BU, значение в поле price увеличилось на 10%:

```
title_id      price
----------    -----------
BU1032        21.9890
BU1111        13.1450
BU2075        3.2890
BU7832        21.9890

(4 row(s) affected)
```

На этом возможности манипулирования результатами далеко не исчерпываются. Например, предположим, что компания руководствуется бизнес-правилом, согласно которому товары должны иметь цены, которые могут быть оплачены без остатка с помощью денежных знаков, имеющих хождение в США. Цены, сформированные после увеличения их на 10%, не соответствуют указанному критерию, поэтому нужно что-то сделать для их округления до ближайшего значения, выраженного в целых центах (0,01 доллара). Можно было бы продолжить рассматриваемый вариант и округлить цены до ближайшего целого цента, выполнив еще одну операцию обновления, в которой осуществляется округление, но лучше вернемся к началу. Прежде всего отменим последнюю операцию обновления:

```
UPDATE titles
SET price = price / 1.1
WHERE title_id LIKE 'BU%'
```

Обратите внимание на то, что для этого достаточно было откорректировать только одну строку кода. После выполнения этого действия тот же оператор SELECT должен показать, что результаты снова стали такими же, с которых мы начинали:

```
title_id      price
----------    --------
BU1032        19.9900
BU1111        11.9500
BU2075        2.9900
BU7832        19.9900

(4 row(s) affected)
```

Теперь мы можем применить более усовершенствованный запрос с самого начала. На этот раз будет выполнено в основном такое же обновление, но с учетом округления обновляемых данных:

```
UPDATE titles
SET price = ROUND(price * 1.1, 2)
WHERE title_id LIKE 'BU%'
```

Фактически до того как с помощью оператора UPDATE вносятся изменения в каждую строку, выполняются две математические операции. Вначале происходит такая

же операция, как и в первом запросе (цена увеличивается на 10%). Затем эта промежуточная цена округляется в соответствии с указанным бизнес-правилом (согласно которому цена должна определяться с точностью до цента). Для этого используется функция ROUND(), параметр которой указывает, что данные должны округляться до двух десятичных позиций (на это указывает цифра 2, стоящая после коэффициента 1.1, обеспечивающего увеличение на 10%). Замечательной особенностью этого оператора является то, что он позволяет выполнить все необходимые действия с помощью одной операции, а не двух.

Проверим полученный результат:

```
title_id         price
-------------    ----------
BU1032           21.9900
BU1111           13.1500
BU2075           3.2900
BU7832           21.9900

(4 row(s) affected)
```

Вполне очевидно, что даже простой оператор UPDATE может оказаться довольно мощным. Но на этом его возможности далеко не исчерпываются. Более сложные варианты обновления рассматриваются в последующих главах.

> Безусловно, СУБД SQL Server предоставляет пользователю настолько большие удобства, что возможно даже обновление значений практически любого столбца (число исключений из этого правила невелико; например, нельзя обновлять столбцы с временными отметками), но при обновлении первичных ключей необходимо соблюдать исключительную осторожность. Обновление ключей связано с очень высоким риском зависания, т.е. потери связи с исходными данными других данных (ссылающихся на те данные, которые подвергаются изменениям).
>
> Например, столбец stor_id в таблице stores базы данных pubs представляет собой столбец первичного ключа. Если бы было принято решение изменить значение 10 в столбце stor_id таблицы stores на 35, то все данные в таблице sales, которые относятся к складу с идентификатором 10, могут стать зависшими и недоступными, если значение stor_id во всех строках, ссылающихся на значение 10 поля stor_id, не станет также равным 35 в результате этого обновления. Тем не менее в базе данных pubs предусмотрено ограничение целостности, касающееся применения ссылок на таблицу stores, поэтому в данном случае СУБД SQL Server сама следит за тем, чтобы не возникали ситуации, связанные с зависанием (дополнительные сведения об ограничениях целостности приведены в главе 7), но еще раз отметим, что могут быть случаи, когда защита первичных ключей от обновления не предусмотрена, а это очень опасно.

Оператор DELETE

По-видимому, самым простым из всех операторов, рассматриваемых в данной главе, является описанная в этом разделе версия оператора DELETE. В этой версии не предусмотрено использование списка столбцов, а задается только имя таблицы и, как правило, конструкция WHERE. Поэтому трудно представить себе более простой синтаксис:

```
DELETE <table_name>
[WHERE <search condition>]
```

Конструкция WHERE действует точно так же, как и все конструкции WHERE, рассматривавшиеся до сих пор. Список столбцов задавать не требуется, поскольку удаляется целая строка (например, невозможно удалить половину строки).

Поскольку изучение оператора DELETE не должно быть сопряжено с какими-либо сложностями, в настоящем разделе будет приведен только ряд небольших примеров операторов удаления, предназначенных для уничтожения следов тех операций вставки, которые были выполнены ранее в данной главе. Вначале вызовем на выполнение оператор SELECT, чтобы убедиться, что первая из указанных строк все еще находится в таблице:

```
SELECT *
FROM stores
WHERE stor_id = 'TEST'
```

Если эта строка еще не удалена, то результаты выполнения предыдущего оператора должны показать наличие одной строки, совпадающей с той, которая была первоначально добавлена к таблице с помощью оператора INSERT. Теперь избавимся от этой строки:

```
DELETE stores
WHERE stor_id = 'TEST'
```

Обратите внимание на то, что в данном случае мы столкнемся с ситуацией, в которой СУБД SQL Server откажется удалить строку в связи с нарушениями ссылочной целостности:

```
Msg 547, Level 16, State 1, Line 1
DELETE statement conflicted with COLUMN REFERENCE constraint
'FK__sales__stor_id__1BFD2C07'. The conflict occurred in database 'pubs',
table 'sales', column 'stor_id'.
The statement has been terminated.
```

СУБД SQL Server не позволяет удалять строку, если на нее имеется ссылка, на которую распространяется ограничение внешнего ключа. Дополнительные сведения о внешних ключах будут приведены в главе 7, но на данный момент достаточно указать, что если одна строка ссылается на другую строку (либо в той же, либо в другой таблице; это не имеет значения) с использованием внешнего ключа, то необходимо вначале удалить ссылающуюся строку и только затем удалить строку, на которую была сделана ссылка. В последнем операторе INSERT в таблицу sales была вставлена строка со значением stor_id, равным TEST, а именно эта строка ссылается на строку, которую мы только что пытались удалить.

Поэтому, прежде чем удалить указанную строку из таблицы stores, необходимо удалить строку таблицы sales, которая на нее ссылается:

```
DELETE sales
WHERE stor_id = 'TEST'
```

Теперь можно повторно вызвать на выполнение первый оператор DELETE, который на сей раз будет выполнен успешно:

```
DELETE stores
WHERE stor_id = 'TEST'
```

Чтобы убедиться в том, что данные действительно удалены, можно провести две простые проверки. Первая из них происходит автоматически при выполнении опера-

тора DELETE. Ее успешное проведение подтверждает сообщение о том, что воздействию оператора подверглась одна строка. Вторая несложная проверка состоит в повторном вызове на выполнение оператора SELECT. При этом должно быть получено нуль строк.

Для проведения еще одного простого практического занятия по использованию оператора DELETE уничтожим также вторую из вставленных ранее строк, внеся в текст оператора лишь небольшое изменение:

```
DELETE stores
WHERE stor_id = 'TST2'
```

На этом тема, связанная с выполнением простых операторов удаления, заканчивается. Оператор DELETE, как и другие операторы, которые рассматривались в настоящей главе, станет темой дальнейшего изложения после того, как мы будем готовы приступить к использованию более сложных условий поиска.

Резюме

Язык T-SQL – это разновидность языка SQL (Structured Query Language – язык структурированных запросов), определяемого стандартом ANSI, которая применяется исключительно в СУБД Server SQL. Язык T-SQL совместим со стандартом ANSI 92 в минимальной конфигурации, а также включает целый ряд собственных расширений, о чем более подробно сказано в последующих главах.

В СУБД SQL Server в целях обеспечения обратной совместимости допускается использование множества различных вариантов синтаксиса, которые фактически не отличаются по своим возможностям от синтаксиса, соответствующего стандарту ANSI, но везде, где это возможно, следует использовать форму ANSI. Как правило, в данной книге при наличии нескольких вариантов синтаксического оформления операторов демонстрируются все варианты, но опять-таки по возможности применяется вариант ANSI. Это особенно важно, если в дальнейшем в какой-то момент может возникнуть необходимость поменять серверную часть приложения (иными словами, перейти к использованию другого сервера базы данных). Вероятность того, что код ANSI будет успешно функционировать на новом сервере базы данных, весьма велика, тогда как код, характеризующийся использованием только средств T-SQL, потребует существенной переработки.

В настоящей главе подробно описаны операторы T-SQL, предназначенные для обработки отдельных таблиц, но реальность такова, что чаще всего требуется информация больше чем из одной таблицы. В следующей главе будет показано, как использовать операции JOIN для выборки данных из нескольких таблиц.

Упражнения

3.1. Напишите запрос, который выводит все столбцы и все строки из таблицы authors базы данных pubs.

3.2. Измените запрос, рассматриваемый в упражнении 1, таким образом, чтобы в полученных результатах остались сведения только об авторах из штата Юта (Utah). (Подсказка: их должно быть 2.)

3.3. Добавьте новую строку в таблицу authors базы данных pubs.

3.4. Удалите только что добавленную строку.

Соединения

С этой главы начинается настоящая профессиональная подготовка по языку SQL. Операторы, которые рассматривались в предыдущих главах, составляют лишь небольшую часть инструментария разработчика, поскольку они в основном предназначены для работы с отдельными таблицами, что не позволяет выполнить достаточно большой объем работы, особенно если база данных в значительной степени нормализована.

Нормализованная база данных характеризуется тем, что в ней данные не концентрируются в больших, всеобъемлющих таблицах, а распределяются по многочисленным таблицам с меньшим количеством столбцов в целях устранения повторяющихся данных, экономии пространства, повышения производительности и обеспечения целостности данных. Задача нормализации является очень важной и крайне необходимой для обеспечения нормального функционирования реляционных баз данных; тем не менее, из того, что данные нормализованы, т.е. распределены по многочисленным таблицам, следует также, что для выборки данных приходится использовать несколько таблиц.

> *Концепции нормализации рассматриваются более подробно в главе 8. А на данный момент достаточно помнить, что по мере увеличения степени нормализации базы данных возрастает вероятность того, что для получения всех требуемых данных придется прибегать к соединению нескольких таблиц.*

В настоящей главе приведено вводное описание процесса комбинирования данных из нескольких таблиц в один результирующий набор с использованием различных форм конструкции `JOIN`. В число этих конструкций входят следующие:

- `INNER JOIN` — внутреннее соединение.
- `OUTER JOIN` — внешнее соединение, как левое (`LEFT`), так и правое (`RIGHT`).
- `FULL JOIN` — полное соединение.
- `CROSS JOIN` — перекрестное соединение.

В этой главе будет также показано, что существует несколько вариантов синтаксиса операторов соединения и наиболее подходящим является один конкретный вариант синтаксиса. Кроме того, будет кратко представлена операция UNION, позволяющая объединять результаты двух запросов в один результирующий набор.

Конструкции JOIN

Во время работы с данными, хранящимися в нормализованной базе данных, часто приходится сталкиваться с ситуациями, в которых всю необходимую информацию невозможно получить только из одной таблицы. В других случаях вся информация, которая должна быть получена, находится в одной таблице, но выборка этих данных должна осуществляться по условиям, подлежащим проверке по другой таблице. Конструкция JOIN предназначена для использования именно в таких ситуациях.

Конструкция JOIN выполняет именно ту задачу, на которую указывает смысл соответствующего английского глагола, — соединяет информацию из двух таблиц в один результирующий набор. Результирующий набор можно рассматривать как "виртуальную" таблицу. В него входят и столбцы, и строки, а сами столбцы характеризуются определенными типами данных. И действительно, в главе 7 будет показано, что результирующий набор можно использовать так, как если бы это была таблица, и обращаться к нему для выполнения других запросов.

Конкретные способы, применяемые в конструкции JOIN для соединения информации из двух таблиц в один результирующий набор, зависят от того, какие указания предусмотрены в этой конструкции, касающиеся сбора данных из разных таблиц, поэтому и предусмотрены четыре разных типа конструкции JOIN. Но все разновидности конструкций JOIN имеют одну общую отличительную особенность в том, что в них одна строка согласуется с одной или несколькими другими строками для получения результирующей строки, представляющей собой надмножество, созданное путем соединения полей из нескольких записей.

Например, предположим, что строки с данными о кинофильмах берутся из таблицы Films (табл. 4.1).

Таблица 4.1. Одна строка с данными из таблицы Films

FilmID	FilmName	YearMade
1	My Fair Lady	1964

Теперь перейдем к рассмотрению строки из таблицы с данными об актерах, называемой Actors (табл. 4.2).

Таблица 4.2. Одна строка с данными из таблицы Actors

FilmID	FirstName	LastName
1	Rex	Harrison

Конструкция JOIN позволяет создать одну строку из двух строк, находящихся в полностью отдельных таблицах (табл. 4.3).

Таблица 4.3. Строка, полученная в результате соединения строк с данными из таблиц Films и Actors

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison

С помощью этой конструкции JOIN строки соединяются на основании связи "один к одному" (по крайней мере такое впечатление складывается на основании приведенных данных). Одна строка из таблицы Films соединяется с одной строкой из таблицы Actors.

Немного дополним условия этого примера и рассмотрим, что при этом произойдет. Введем еще одну строку в таблицу Actors (табл. 4.4).

Таблица 4.4. Дополненная таблица Actors

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn

Теперь рассмотрим, что произойдет после соединения дополненной таблицы Actors с той же таблицей Films (содержащей только одну строку) (табл. 4.5).

Таблица 4.5. Результаты соединения дополненной таблицы Actors с таблицей Films

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn

Вполне очевидно, что полученные данные существенно изменились, поскольку больше нельзя утверждать, что между таблицами наблюдается связь "один к одному"; скорее, здесь присутствует связь "один к двум", вернее, связь, которую с большим основанием можно назвать "один ко многим". Одна-единственная строка из таблицы Films может использоваться столько раз, сколько потребуется для получения полных (соединенных) данных из согласующихся строк таблицы Actors.

Обратил ли читатель внимание на то, как происходит согласование? Безусловно, строки согласуются путем проверки значений поля FilmID из обеих таблиц для создания одной строки из двух.

Но в рассматриваемых примерах используются настолько ограниченные наборы данных, что полученные результаты практически не должны зависеть от применяемой конструкции JOIN. Тем не менее разные варианты этой конструкции обладают своими особенностями, которые рассматриваются в следующих разделах.

Конструкции INNER JOIN

Бесспорно, конструкции INNER JOIN представляют собой наиболее распространенную разновидность JOIN. С помощью этих конструкций осуществляется согласование строк по данным из одного или нескольких общих полей, как и в большинстве других конструкций JOIN, но в отличие от этого конструкция INNER JOIN возвращает только строки, согласованные по всем полям, которые обозначены как используемые

для соединения. В предыдущих примерах в результирующий набор вошла по меньшей мере один раз каждая строка, но на практике такая ситуация встречается редко.

Дополним рассматриваемые таблицы и проверим, какие результаты могут быть получены с помощью конструкции INNER JOIN. Допустим, что теперь таблица Films имеет такой вид, как показано в табл. 4.6.

Таблица 4.6. Дополненная таблица Films

FilmID	FilmName	YearMade
1	My Fair Lady	1964
2	Unforgiven	1992

А таблица Actors стала выглядеть так, как показано в табл. 4.7.

Таблица 4.7. Таблица Actors после еще одного дополнения

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn
2	Clint	Eastwood
5	Humphrey	Bogart

После использования операции выборки с конструкцией INNER JOIN результирующий набор принимает такой вид, как показано в табл. 4.8.

Таблица 4.8. Результирующий набор, полученный с применением конструкции INNER JOIN

FilmID	FilmName	YearMade	FirstName	LastName
1	My Fair Lady	1964	Rex	Harrison
1	My Fair Lady	1964	Audrey	Hepburn
2	Unforgiven	1992	Clint	Eastwood

Обратите внимание на то, что в этом результирующем наборе не встречается фамилия актера Bogey. Это связано с тем, что соответствующая строка в таблице Films отсутствует. В результирующий набор не включается строка, для которой отсутствуют согласующиеся поля в обеих таблицах. Теперь перейдем к рассмотрению самих примеров кода.

Наиболее предпочтительный формат кода для конструкции INNER JOIN выглядит примерно таким образом:

```
SELECT <select list>
FROM <first_table>
<join_type> <second_table>
      [ON <join_condition>]
```

Это – синтаксис, предусмотренный стандартом ANSI. Он в основном предназначен для использования в системах баз данных, отличных от SQL Server. С другой стороны, в версиях СУБД SQL Server вплоть до 6.0 и предшествующих ей версий необходимо было использовать собственный синтаксис (который до сих пор еще применяют многие разработчики). Такая специальная разновидность синтаксиса будет рассматриваться ниже в данной главе.

Вызовите на выполнение программу Management Studio и проведите испытания операторов с конструкцией INNER JOIN, применив следующий код по отношению к базе данных Northwind:

```
SELECT *
FROM Products
INNER JOIN Suppliers
      ON Products.SupplierID = Suppliers.SupplierID
```

Результаты выполнения этого запроса занимают слишком много места по ширине, поэтому не приведены в настоящей книге, но после выполнения данного запроса должно быть получено примерно 77 строк. Эти результаты отличаются несколькими особенностями, перечисленными ниже.

- Столбец SupplierID встречается дважды, но не имеет каких-либо признаков, позволяющих узнать, какой из этих столбцов относится к той или иной таблице.
- Происходит возврат всех столбцов из обеих таблиц.
- Первым столбцом, вошедшим в результирующий набор, является столбец из первой указанной таблицы.

Можно было бы узнать, к чему относится идентификатор SupplierID, просто рассматривая таблицу, выбранную в первую очередь, и согласуя данные из этого столбца с первым обнаруженным столбцом SupplierID. Тем не менее такая процедура является утомительной, а что еще хуже, способствует возникновению ошибок. В этом состоит одна из многих причин, по которым не рекомендуется указывать в конструкциях JOIN простой подстановочный символ * вместо списка столбцов. Однако в случае применения конструкции INNER JOIN указанная проблема не является столь существенной, поскольку известно, что оба столбца SupplierID должны быть точными дубликатами по отношению друг к другу, даже несмотря на то, что берутся из разных таблиц. Чтобы убедиться в этом, достаточно вспомнить, что в конструкции INNER JOIN применяются два указанных столбца, поэтому данные в них должны согласовываться, иначе не будет возвращена ни одна строка. Но на это нельзя постоянно рассчитывать. На основании изучения соединений других типов можно убедиться в том, что нельзя полагаться на то, что согласуемые в конструкции JOIN значения всегда должны быть равны.

А что касается того, что приведенный выше оператор возвращает все столбцы из обеих таблиц, то такой результат вполне соответствует ожиданиям. В операторе выборки использовался подстановочный символ *, который, как уже было сказано, обеспечивает получение данных из всех столбцов. Однако, как уже отмечалось, применение подстановочного символа * в операторах соединения не рекомендуется. Операторы, имеющие подобный формат, являются краткими и удобными, но вместе с тем обладают существенными недостатками, поскольку их применение может повлечь за собой возникновение ошибок и способствовать снижению производительности.

Одна из обоснованных рекомендаций, которую следует принять на вооружение с самого начала изучения программирования на языке SQL, состоит в том, чтобы осуществлять выборку только необходимой информации и действительно использовать в программе данные, полученные в результате выборки. Эта рекомендация основана на том, что для передачи любых дополнительных данных строки или столбца требуется дополнительная пропускная способность сети, а также чаще всего дополнительная обработка запроса в СУБД SQL Server. В конечном итоге обнаруживается, что выборка ненужной информации отрицательно вли-

яет на производительность работы не только того пользователя, который в настоящий момент работает с программой, но и всех других пользователей системы, а также пользователей сети, в которой находится СУБД SQL Server.

Указывайте в операторе выборки только те столбцы, данные из которых вы собираетесь использовать, и применяйте настолько ограничительные конструкции WHERE, насколько это возможно.

Если вам обязательно требуется применять подстановочный символ *, то при этом необходимо ограничиваться только теми таблицами, из которых должны быть получены данные всех столбцов. В таком случае использование подстановочного символа * будет вполне оправданным, поскольку никто не отрицает возможности выборки с его помощью данных из отдельных таблиц. Например, если бы требовалась вся информация о товарах, а из таблицы с данными о поставщиках нужно было получить только название компании поставщика, то можно было бы модифицировать приведенный выше запрос следующим образом:

```
SELECT Products.*, CompanyName
FROM Products
INNER JOIN Suppliers
      ON Products.SupplierID = Suppliers.SupplierID
```

Прокрутка результатов этого запроса на экране слева направо показывает, что основная часть информации о поставщиках теперь исключена. Безусловно, остался также только один экземпляр столбца SupplierID. В конечном итоге в полученный результирующий набор вошли все столбцы из таблицы Products (поскольку подстановочный символ * был указан только для этой таблицы, а единственный экземпляр столбца SupplierID поступил из этой части списка выборки); с другой стороны, присутствует отдельный столбец с именем CompanyName (который берется из таблицы Suppliers). Теперь еще раз попытаемся выполнить тот же запрос, внеся в него только одно небольшое изменение:

```
SELECT Products.*, SupplierID
FROM Products
INNER JOIN Suppliers
      ON Products.SupplierID = Suppliers.SupplierID
```

Но на сей раз возникла проблема — появилось следующее сообщение об ошибке:

```
Msg 209, Level 16, State 1, Line 1
Ambiguous column name 'SupplierID'.
```

Почему применение в запросе столбца CompanyName рассматривается как допустимое, а столбца SupplierID — нет? Именно по той причине, которая указана в сообщении СУБД SQL Server, — применяемое имя столбца является неоднозначным. Столбец CompanyName имеется только в таблице Suppliers, а столбец SupplierID присутствует в обеих таблицах, поэтому СУБД SQL Server не может определить, какой именно столбец нам требуется. Во всех вариантах запросов, применявшихся до сих пор для получения данных SupplierID, задача идентификации этого столбца была разрешимой. Иными словами, СУБД SQL Server получала указания, позволяющие определить, из какой таблицы должны быть взяты данные этого столбца. В первом запросе (в котором использовался простой подстановочный символ *) СУБД SQL Server передавалось задание возвратить всю доступную информацию; в состав этой информации должны были войти оба столбца SupplierID, поэтому определение того, к какой таблице относят-

ся имена столбцов, не требовалось. Во втором примере (в котором было указано, что действие подстановочного символа * распространяется только на таблицу Products) снова не было дано каких-либо конкретных указаний об использовании того или иного столбца SupplierID. Вместо этого была предусмотрена выборка данных всех столбцов из таблицы Products, причем столбец SupplierID просто оказался одним из столбцов, вошедших в этот список. А задача определения того, к чему относится столбец CompanyName, оказалась легко разрешимой, поскольку в двух таблицах есть только один столбец CompanyName, поэтому именно он нам и требуется.

Таким образом, если необходимо сослаться на такой столбец, что в результатах соединения с помощью конструкции JOIN должно появиться несколько столбцов с одинаковыми именами, то требуется **полностью уточнить** имя столбца. Для этого может применяться один из двух описанных ниже способов.

- Указать имя таблицы, в которой находится требуемый столбец, затем поставить точку и указать имя столбца (например, Table.ColumnName).
- Обозначить таблицу псевдонимом и указать этот псевдоним, затем поставить точку и указать имя столбца (например, Alias.ColumnName).

Задача полного уточнения имен является несложной; пример того, как она решается, мы уже видели применительно к уточненному подстановочному символу *, однако еще раз проверим приведенный выше запрос, относящийся к столбцу SupplierID, но с полностью уточненным именем столбца:

```
SELECT Products.*, Suppliers.SupplierID
FROM Products
INNER JOIN Suppliers
      ON Products.SupplierID = Suppliers.SupplierID
```

После этого возобновляется нормальная работа оператора выборки и в крайней правой части результирующего набора появляются данные столбца SupplierID из таблицы Suppliers.

С другой стороны, способ, в котором предусматривается создание псевдонима для таблицы, лишь ненамного сложнее, но позволяет добиться того, чтобы формулировка запроса стала более лаконичной, а его удобство для чтения повысилось. Этот способ реализуется почти точно так же, как и способ применения псевдонима для столбца в простых операторах SELECT, которые рассматривались в предыдущей главе. При этом достаточно лишь ввести сразу вслед за именем таблицы тот псевдоним, который должен использоваться для ссылок на эту таблицу. Следует отметить, что, как и при указании псевдонимов столбцов, можно использовать ключевое слово AS (но по какой-то странной причине это ключевое слово не нашло широкого распространения на практике):

```
SELECT p.*, s.SupplierID
FROM Products p
INNER JOIN Suppliers s
      ON p.SupplierID = s.SupplierID
```

После вызова этого кода на выполнение можно обнаружить, что полученные результаты полностью совпадают с теми результатами, которые были сформированы в предыдущем запросе.

Следует отметить, что псевдонимы применяются по принципу "все или ничего". После принятия решения об использовании псевдонима для какой-то таблицы этот

псевдоним должен быть подставлен вместо имени таблицы во всех частях запроса. С другой стороны, для одних таблиц могут быть заданы псевдонимы, а для других — нет, но в таком случае код теряет единообразие. В этом можно убедиться, ознакомившись со следующим запросом:

```
SELECT p.*, Suppliers.SupplierID
FROM Products p
INNER JOIN Suppliers s
      ON p.SupplierID = s.SupplierID
```

На первый взгляд может показаться, что этот запрос должен быть выполнен успешно, но он приводит к возникновению следующей ошибки:

```
Msg 4104, Level 16, State 1, Line 1
The multi-part identifier "Suppliers.SupplierID" could not be bound.
```

Еще раз отметим, что ситуация, в которой для одних таблиц предусмотрены псевдонимы, а для других — нет, является допустимой, но после принятия решения об использовании псевдонима для некоторой таблицы необходимо неизменно придерживаться выбранного обозначения.

Еще раз вернемся к тем примерам, которые рассматривались в начале данной главы; в них заслуживает внимания еще одна особенность, которая состоит в том, что в первую очередь происходит возврат столбцов из первой таблицы, указанной в операторе с конструкцией JOIN. Сделаем небольшое отступление и рассмотрим, с чем это связано и какие действия следует предпринять, чтобы получить возможность управлять последовательностью расположения столбцов.

В СУБД SQL Server всегда используется такой порядок столбцов, который выбран на основании наиболее оправданных предположений в отношении того, в каком порядке хотел бы получить эти столбцы сам пользователь. В первом запросе использовался универсальный подстановочный символ *, поэтому для СУБД SQL Server не оставалось достаточно большого выбора. В данном случае остается учесть при выборе такие два фактора: порядок столбцов, в соответствии с которым эти столбцы физически представлены в таблице, и порядок, в котором таблицы указаны в запросе. При этом весьма удобно то, что задача переупорядочения столбцов является чрезвычайно простой, — достаточно просто явно задать эти столбцы. Еще более простым является способ изменения порядка следования столбцов, согласно которому изменяется последовательность указания самих таблиц, но фактически мы можем изменять порядок столбцов как угодно, явно указывая, какие столбцы должны использоваться для формирования результирующего набора (даже если при этом потребуется указать каждый столбец), а также задать порядок, в котором должны быть представлены эти данные.

Практическое занятие

Применение простой конструкции JOIN

Рассмотрим небольшой запрос, который позволяет ознакомиться с особенностями конструкции JOIN:

```
SELECT p.ProductID, s.SupplierID, p.ProductName, s.CompanyName
FROM Products p
INNER JOIN Suppliers s
      ON p.SupplierID = s.SupplierID
WHERE p.ProductID < 4
```

Выполнение этого запроса приводит к получению довольно простого результирующего набора:

```
ProductID   SupplierID   ProductName          CompanyName
----------- ------------ -------------------- ----------------
1           1            Chai                 Exotic Liquids
2           1            Chang                Exotic Liquids
3           1            Aniseed Syrup        Exotic Liquids

(3 row(s) affected)
```

Описание полученных результатов

В отличие от тех примеров, в которых не были даны конкретные указания в отношении того, какие столбцы должны присутствовать в полученных результатах (поскольку использовался просто подстановочный символ *), на этот раз необходимые столбцы указаны конкретно, поэтому СУБД SQL Server получила возможность однозначно определить, что нам требуется, — данные столбцов выведены точно в таком же порядке, который указан в списке выборки.

Общие свойства конструкции INNER JOIN и конструкции WHERE

До сих пор при описании особенностей конструкции INNER JOIN фактически затрагивались только те концепции, которые применимы к соединениям любых других типов, поскольку принципы определения порядка расположения столбцов в результирующем наборе и применения псевдонимов являются полностью одинаковыми для конструкций JOIN любых типов. А то, в чем конструкция INNER JOIN отличается от конструкций JOIN других типов, является создание с ее помощью **исключительного соединения**, т.е. соединения, в котором исключены все строки, не имеющие определенного значения в обеих таблицах (в *левой таблице*, как называют таблицу, указанную в первую очередь, и в *правой таблице*, заданной во вторую очередь).

Первый пример проявления этого свойства можно обнаружить по результатам применения конструкции INNER JOIN к воображаемым таблицам Films и Actors. В результаты не вошла строка с данными об актере Bogey, поскольку в таблице Films не был указан соответствующий фильм с участием этого актера. А теперь рассмотрим несколько более реальных примеров того, как проявляется это свойство.

Допустим, что имеется таблица Customers, в которой собраны все данные об именах и адресах заказчиков. Но наличие большого списка заказчиков компании отнюдь не означает, что в текущий момент компанией действительно выполняются заказы, полученные от всех заказчиков. Фактически можно смело предположить, что в текущий момент имеются такие заказчики, которые не разместили в компании ни одного заказа. Запросы предназначены для выборки требуемой информации. В начале данной главы рассматривались такие информационные потребности, для удовлетворения которых требуется применение в запросе конструкций INNER JOIN, а в этом разделе будет показано, что даже небольшое изменение характера искомой информации влечет за собой необходимость перейти к использованию конструкций JOIN других типов.

Например, коммерческий директор может задать администратору базы данных такой вопрос: "Можете ли вы предоставить информацию обо всех заказчиках, заказы которых выполняются в настоящее время?"

На этот вопрос можно смело ответить: "Разумеется!", после чего приступить к проектированию структуры будущего запроса. Итак, что нам требуется? Коммерческий директор задал вопрос, касающийся и заказчиков, и заказов, поэтому примем предположение, что для получения этой информации должны использоваться обе таблицы. Далее, коммерческому директору требуется только список заказчиков, поэтому в запросе должен быть упомянут столбец `CompanyName`, а также, возможно, столбец `CustomerID`. Следует отметить, что, безусловно, потребуется включить в запрос и таблицу `Orders`, чтобы можно было узнать, было ли что-либо заказано тем или другим заказчиком или нет, но из этой таблицы не требуется возвращать какие-либо данные для дальнейшего использования (именно поэтому в приведенном ниже операторе эта таблица отсутствует в списке выборки). В вопросе коммерческого директора речь идет о том, что должен быть получен список заказчиков, разместивших в компании свои заказы, поэтому для ответа на данный вопрос требуется найти решение, в котором рассматривались бы и строки таблицы `Customers`, и строки таблицы `Orders`. Как уже было сказано, конструкция `INNER JOIN` применяется именно для этой цели. Таким образом, мы выяснили все, что необходимо для составления запроса, и можем сформулировать его следующим образом:

```
SELECT DISTINCT c.CustomerID, c.CompanyName
FROM Customers c
INNER JOIN Orders o
      ON c.CustomerID = o.CustomerID
```

Если в данные базы данных `Northwind` не были внесены изменения, то выполнение этого запроса должно привести к получению 89 строк. Обратите внимание на то, что в запросе используется ключевое слово `DISTINCT`, поскольку достаточно знать только количество заказчиков, сделавших заказы (причем достаточно только одного упоминания каждого заказчика), а не количество заказов. Если бы в этом запросе отсутствовало ключевое слово `DISTINCT`, то была бы возвращена отдельная строка, относящаяся к каждому заказчику для каждой строки из таблицы `Orders`, в которой имеется информация об этом заказчике.

Теперь попытаемся получить информацию об общем количестве заказчиков и для этого вызовем на выполнение следующий простой запрос с агрегирующей функцией `COUNT`:

```
SELECT COUNT(*) AS "No. Of Records" FROM Customers
```

Очевидно, что этот запрос, в котором определяется количество строк в таблице `Customers`, возвращает другие данные по сравнению с предыдущими:

```
No. Of Records
--------------------
91

(1 row(s) affected)
```

С чем связано то, что количество строк отличается на два? Как и следовало ожидать, в первом случае из результирующего набора были исключены две строки с информацией о заказчиках, поскольку данным об этих заказчиках не соответствует ни

одна строка в таблице Orders. Именно по этой причине конструкцию INNER JOIN иногда сравнивают с конструкцией WHERE. Применение конструкции INNER JOIN приводит к исключению строк в связи с тем, что не обнаруживаются соответствующие им строки в другой таблице, а использование конструкции WHERE приводит к исключению строк из возвращаемого набора, поскольку эти строки не соответствуют заданным критериям.

Чтобы лучше изучить свойства соединений и получить дополнительную практику, рассмотрим, какие столбцы имеются еще в нескольких таблицах базы данных pubs (authors, titles и titleauthor) (табл. 4.9).

Таблица 4.9. Столбцы таблиц authors, titles и titleauthor

authors	titles	titleauthor
au_id	title_id	au_id
au_lname	Title	title_id
au_fname	Type	au_ord
Phone	pub_id	royaltyper
address	Price	
City	Advance	
State	Royalty	
Zip	ytd_sales	
contract	notes	
	pubdate	

На этот раз нам предстоит задача составить запрос, который возвращает фамилии всех авторов, написавших книги, и названия написанных ими книг. Рекомендуем читателю потратить определенное время и попытаться подготовить такой запрос самостоятельно и только после этого переходить к изложенному ниже анализу.

Прежде всего необходимо точно выяснить, какие данные должны быть получены из базы данных. В рассматриваемом задании речь идет о том, что должны быть возвращены два различных фрагмента информации — имена авторов и названия книг. Имена авторов (состоящие из двух частей — само имя и фамилия) можно получить из таблицы authors, а названия книг приведены в таблице titles, поэтому мы можем приступить к написанию первой части оператора SELECT следующим образом:

```
SELECT au_lname + ', ' + au_fname AS "Author", title
```

Как и во многих языках, в языке SQL операция "+" может использоваться как для конкатенации строк, так и для сложения чисел. В данном случае просто выполняется конкатенация строки фамилии со строкой имени и между этими двумя строками ставится разделитель в виде запятой.

Теперь необходимо приступить к соединению двух указанных таблиц. Но для этого требуется найти столбец, по которому должно быть выполнено соединение. В связи с этим в ходе решения данной задачи возникает первая проблема — обнаруживается, что такой столбец отсутствует. Оказывается, что в этих двух таблицах нет общих столбцов, на которых можно было бы сформировать соединение с помощью конструкции JOIN.

Теперь становится ясно, для чего нужна третья из таблиц, приведенных в табл. 4.9. Иногда соединения могут быть сформированы только с помощью такой таблицы, как titleauthor, которая позволяет проложить связь между двумя другими таблицами. Для обозначения подобных таблиц, обеспечивающих возможность связать две другие таблицы, применяются разные термины, но автору чаще всего приходилось встречать термины **связующая таблица**, или **таблица ассоциации**.

> *Связующей таблицей* (иногда называемой также *таблицей ассоциации*, или *таблицей слияния*) называют любую таблицу, основным назначением которой является не хранение собственных данных, а создание связей между данными, хранимыми в других таблицах. Такие таблицы можно рассматривать как средства "обеспечения взаимодействия", или "создания связей" между двумя или несколькими таблицами. В частности, связующие таблицы позволяют найти выход в такой часто складывающейся ситуации, когда имеет место так называемая связь "многие ко многим" между таблицами. В такой ситуации две таблицы содержат связанные друг с другом данные, причем и в той и в другой таблице может находиться большое количество строк, которые согласуются со многими строками в другой таблице. СУБД SQL Server не позволяет непосредственно реализовывать подобные связи, поэтому применяются связующие таблицы, позволяющие разделить связь "многие ко многим" на две связи "один ко многим", а последние поддерживаются СУБД SQL Server. Дополнительная информация на эту тему приведена в главе 8.

Данная конкретная таблица, titleauthor, не отвечает всем критериям определения связующей таблицы в самом строгом смысле этого термина, но все же соответствует общему назначению связующих таблиц, поэтому автор рассматривает ее именно как таковую, поскольку не считает себя педантом. Применение указанной третьей таблицы, titleauthor, позволяет косвенно соединить таблицы authors и titles, формируя соединения между каждой из этих таблиц и связующей таблицей. Соединение между таблицами authors и titleauthor формируется на основе столбца au_id, а соединение между таблицами titles и titleauthor — на основе столбца title_id.

Практическое занятие

Более сложные операторы с конструкциями JOIN

Введение указанной третьей таблицы в конструкции JOIN не составляет труда; для этого достаточно снова указать в конструкции FROM таблицу, в которой находится требуемая информация, и задать ключевые слова JOIN (прежде чем вызвать на выполнение этот оператор, не забудьте переключиться на базу данных pubs):

```
SELECT a.au_lname + ', ' + a.au_fname AS "Author", t.title
FROM authors a
JOIN titleauthor ta
  ON a.au_id = ta.au_id
JOIN titles t
  ON t.title_id = ta.title_id
```

Обратите внимание на то, что таблицам присвоены псевдонимы, поэтому необходимо вернуться в начало оператора и внести изменения в конструкцию SELECT с учетом использования псевдонимов, но на этом составление оператора SELECT с соединением трех таблиц заканчивается! После вызова на выполнение этого оператора будут получены результаты, приведенные в табл. 4.10 (в данном случае применяется режим отображения в виде сетки).

Таблица 4.10. Результаты выполнения оператора соединения

Author	Title
Bennet, Abraham	The Busy Executive's Database Guide
Blotchet-Halls, Reginald	Fifty Years in Buckingham Palace Kitchens
Carson, Cheryl	But Is It User Friendly?
DeFrance, Michel	The Gourmet Microwave
del Castillo, Innes	Silicon Valley Gastronomic Treats
Dull, Ann	Secrets of Silicon Valley
Green, Marjorie	The Busy Executive's Database Guide
Green, Marjorie	You Can Combat Computer Stress!
Gringlesby, Burt	Sushi, Anyone?
Hunter, Sheryl	Secrets of Silicon Valley
Karsen, Livia	Computer Phobic AND Non-Phobic Individuals: Behavior Variations
Locksley, Charlene	Net Etiquette
Locksley, Charlene	Emotional Security: A New Algorithm
MacFeather, Stearns	Cooking with Computers: Surreptitious Balance Sheets
MacFeather, Stearns	Computer Phobic AND Non-Phobic Individuals: Behavior Variations
O'Leary, Michael	Cooking with Computers: Surreptitious Balance Sheets
O'Leary, Michael	Sushi, Anyone?
Panteley, Sylvia	Onions, Leeks, and Garlic: Cooking Secrets of the Mediterranean
Ringer, Albert	Is Anger the Enemy?
Ringer, Albert	Life Without Fear
Ringer, Anne	The Gourmet Microwave
Ringer, Anne	Is Anger the Enemy?
Straight, Dean	Straight Talk About Computers
White, Johnson	Prolonged Data Deprivation: Four Case Studies
Yokomoto, Akiko	Sushi, Anyone?

Следует учитывать, что на другом компьютере в связи с наличием иных критериев сортировки может быть получен отличный от этого порядок строк (напомним, что в СУБД SQL Server не гарантируется возврат результатов в определенном порядке, если не используется конструкция ORDER BY), а поскольку конструкция ORDER BY не предусмотрена, то вступает в силу известная истина: "Фактически применяемый способ представления результатов может измениться".

Описание полученных результатов

Если бы к таблице `authors` был применен простой оператор `SELECT *`, то оказалось бы, что сведения о некоторых авторах не получены, несмотря на то, что сами эти авторы представлены в таблице `authors`. Тем самым создалось бы впечатление, что соответствующие данные о написанных этими авторами книгах отсутствуют (по крайней мере в рассматриваемой базе данных). А в действительности даже при использовании приведенной выше сложной конструкции соединения в итоговые результаты не попало одно название книги (`The Psychology of Computer Cooking`),

поскольку не удалось согласовать это название ни с одним из авторов! Еще раз отметим, что основной особенностью конструкций INNER JOIN является то, что при использовании этих конструкций из результирующего набора исключаются строки, не соответствующие всем заданным критериям.

В отличие от этого, при использовании рассматриваемого оператора сложного соединения вначале формируются результаты соединения второй таблицы с первой, а затем по такому же принципу происходит соединение полученного результирующего набора с третьей таблицей. Подобные операторы можно усложнять, добавляя все новые и новые таблицы, и каждая добавляемая таблица будет действовать по такому же принципу, участвуя в соединении с полученным ранее результирующим набором.

> Обратите внимание на то, что в последнем запросе не используется ключевое слово INNER. Это связано с тем, что соединение INNER JOIN представляет собой применяемый по умолчанию тип соединения JOIN. Принимая решение о том, следует ли явно указывать ключевое слово INNER, разные разработчики придерживаются различных подходов, но сам автор считает, что в силу давно сложившейся практики программирования, согласно которой ключевое слово INNER чаще всего не упоминается, применение конструкции INNER JOIN, а не просто JOIN, скорее даже способствует путанице; именно поэтому в дальнейшем в настоящей книге для обозначения внутреннего соединения будет в основном применяться только ключевое слово JOIN.

Конструкции OUTER JOIN

Применение конструкции JOIN такого типа, как OUTER JOIN, скорее можно считать исключением, а не правилом. Разумеется, причина такой ситуации заключается не в том, что эти конструкции не позволяют выполнять какую-либо полезную работу, а, скорее, в следующем.

- Чаще всего при выборке данных с использованием оператора соединения необходимо обеспечить, чтобы данные соответствовали всем заданным критериям, а этого позволяет добиться только конструкция INNER JOIN.
- Многие разработчики, использующие язык SQL, осваивают лишь внутреннее соединение, осуществляемое с помощью конструкции INNER JOIN, но так и не заходят глубже; иными словами, многие разработчики просто не умеют пользоваться разновидностью оператора соединения с конструкцией OUTER.
- Цели, которые позволяет достичь применение конструкции OUTER JOIN, часто достижимы с помощью других методов.
- Разработчики зачастую просто забывают о том, что может использоваться подобная конструкция.

Еще раз отметим, что при использовании конструкции INNER JOIN исключаются все строки, не соответствующие всем заданным критериям, а при использовании конструкции OUTER, а также, как будет показано ниже в этой главе, конструкции FULL JOIN существует возможность включить в результирующий набор строки, которые соответствуют хотя бы одному из заданных критериев. Иногда с помощью конструкции OUTER JOIN удается легко решить задачи, которые на первый взгляд кажутся очень

сложными, поэтому очень жаль, что многие не умеют применять эти конструкции. Кроме того, с помощью конструкций OUTER JOIN часто удается повысить быстродействие при их использовании вместо вложенных подзапросов (которые будут описаны в главе 7).

Выше в данной главе было указано, что таблицы, участвующие в операции JOIN, подразделяются на находящуюся слева и на находящуюся справа (или просто на левую и правую). Левой считается таблица, указанная в первую очередь, а правой — таблица, указанная после нее. Рассматривая операторы с конструкцией INNER JOIN, мы в основном не придавали этому различию значения, поскольку во внутреннем соединении обе стороны всегда рассматриваются на равных. Но когда речь идет о конструкциях OUTER JOIN, понимание того, что обработка левой таблицы происходит иначе, чем правой, становится буквально необходимым. Впервые сталкиваясь с этим различием, можно подумать, что в нем нет ничего сложного; и действительно, все обстоит очень просто, но слишком часто встречаются ошибки в запросах, основанных на использовании конструкции OUTER JOIN, которые обусловлены тем, что не учитывается различие между левой и правой таблицами.

Чтобы научиться правильно формировать запросы с конструкцией OUTER JOIN, вначале рассмотрим две иллюстрации к использованию синтаксиса. В первом примере рассматривается простой вариант конструкции OUTER JOIN с двумя таблицами, а во втором примере речь идет о более сложном варианте, в котором конструкции OUTER JOIN применяются в сочетании с конструкциями JOIN любых других типов.

Простой вариант оператора с конструкцией OUTER JOIN

Задача освоения первого варианта синтаксиса является несложной, и большинство разработчиков с ней успешно справляются:

```
SELECT <SELECT list>
FROM <the table you want to be the "LEFT" table>
<LEFT|RIGHT> [OUTER] JOIN <table you want to be the "RIGHT" table>
                    ON <join condition>
```

Читатель должен отметить, что в приведенных ниже примерах, как правило, используются полные синтаксические обозначения операций, т.е. обозначения с ключевым словом OUTER (например, LEFT OUTER JOIN). Но следует учитывать, что ключевое слово OUTER является необязательным; достаточно лишь включить ключевое слово LEFT или RIGHT (например, LEFT JOIN).

Автор еще раз хочет заострить внимание читателя на том, что таблица, имя которой упоминается перед ключевым словом JOIN, рассматривается как левая таблица, LEFT, а таблица, имя которой следует за ключевым словом JOIN, — как правая таблица, RIGHT.

Как уже было сказано, применение операторов с конструкцией OUTER JOIN приводит к получению не только тех данных, которые соответствуют всем критериям, но и данных, соответствующих лишь некоторым критериям, а то, какие именно строки, соответствующие лишь некоторым критериям, будут включены в результирующий набор, зависит от выбора той или иной стороны соединения. При использовании конструкции LEFT OUTER JOIN включается вся информация из таблицы, указанной слева от ключевого слова JOIN, а при использовании конструкции RIGHT OUTER JOIN — вся информация из таблицы, указанной справа от этого ключевого слова. Рассмотрим

практические примеры применения запросов с левым и правым внешними соединениями, позволяющие лучше понять сказанное.

Предположим, что необходимо узнать, какие скидки предоставляются покупателю, величину каждой скидки и названия магазинов, в которых эти скидки предоставляются. В базе данных pubs находятся таблицы discounts и stores, которые определены, как показано в табл. 4.11.

Таблица 4.11. Столбцы таблиц discounts и stores

discounts	stores
discounttype	stor_id
stor_id	stor_name
Lowqty	stor_address
Highqty	city
Discount	state
	zip

Эти таблицы имеют общий столбец, stor_id, поэтому можно попытаться непосредственно выполнить их соединение. Оператор, составленный с применением обычной конструкции INNER JOIN, должен выглядеть примерно таким образом:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d
JOIN stores s
  ON d.stor_id = s.stor_id
```

Использование этого оператора приводит к получению только одной строки:

```
discounttype          discount        stor_name
---------------       -------------   ------------
Customer Discount     5.00            Bookbeat

(1 row(s) affected)
```

Однако наше задание отнюдь нельзя назвать выполненным. Нам требовались результаты, в которых содержалась бы информация обо всех возможных скидках, а не только о тех, которые фактически используются. Тем не менее выполнение данного запроса привело к получению только данных о скидке, которая применяется в некотором магазине, а ответ на заданный вопрос не получен!

В действительности требуется такая конструкция, которая позволяет получить сведения обо всех скидках и магазинах, в которых они применяются.

Практическое занятие

Использование конструкции LEFT OUTER JOIN

Для получения сведений о каждой скидке и о тех магазинах, в которых они применяются, достаточно только изменить тип конструкции JOIN в запросе следующим образом:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d
LEFT OUTER JOIN stores s
          ON d.stor_id = s.stor_id
```

Применение данного запроса приводит к получению немного других результатов:

```
discounttype          discount        stor_name
------------------    ------------    ------------
Initial Customer      10.50           NULL
Volume Discount       6.70            NULL
Customer Discount     5.00            Bookbeat

(3 row(s) affected)
```

Попытка выполнения оператора SELECT * применительно к таблице discounts позволяет сразу же обнаружить, что в результаты предыдущего запроса вошли все строки из этой таблицы.

Описание полученных результатов

Применяется операция левого соединения, причем слева от ключевого слова JOIN находится таблица discounts. А как обстоят дела с таблицей stores? Если выполняется соединение и в таблице stores отсутствует строка, соответствующая соединяемой строке таблицы discounts, то что произойдет? В данном случае таблица stores не находится с той стороны от ключевого слова JOIN, которая предусматривает включение всех строк (а именно, с левой стороны), поэтому СУБД SQL Server подставляет NULL вместо любого значения, которое соответствует противоположной (правой) стороне соединения, если отсутствует строка, соответствующая той стороне соединения, в которой включаются все строки (левой). В данном случае в строках, соответствующих всем названиям магазина stor_name, кроме одного, содержатся NULL-значения. На основании этого можно сделать вывод, что для двух строк таблицы discounts (в которых содержатся NULL-значения в столбце, взятом из таблицы stores) нет соответствующих строк в таблице stores, т.е. отсутствуют магазины, использующие скидку указанного типа.

Итак, получен ответ на рассматриваемый вопрос — из трех типов скидок применяется только один (Customer Discount), и этот тип находит свое применение только в одном магазине (Bookbeat).

Практическое занятие

Использование конструкции RIGHT OUTER JOIN

Теперь рассмотрим, что произойдет после перехода к применению соединения типа RIGHT OUTER JOIN:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d
RIGHT OUTER JOIN stores s
           ON d.stor_id = s.stor_id
```

Даже несмотря на то, что эта модификация на первый взгляд кажется весьма незначительной, фактически она приводит к резкому изменению состава результирующего набора:

```
discounttype        discount        stor_name
---------------     -------------   --------------------
NULL                NULL            Eric the Read Books
NULL                NULL            Barnum's
NULL                NULL            News & Brews
NULL                NULL            Doc-U-Mat: Quality Laundry and Books
NULL                NULL            Fricative Bookshop
Customer Discount   5.00            Bookbeat

(6 row(s) affected)
```

Описание полученных результатов

Если теперь оператор SELECT * будет выполнен применительно к таблице stores, то обнаружится, что в состав результатов запроса включены все строки из таблицы stores, причем при наличии соответствующей строки в таблице discounts отображается относящаяся к этой строке информация о скидке. А во всех остальных случаях столбцы, взятые из таблицы discounts, заполняются NULL-значениями. Итак, если допустить, что таблица discounts всегда будет упоминаться в запросе в первую очередь, а таблица stores — во вторую, то, чтобы получить информацию обо всех скидках, нужно использовать конструкцию LEFT JOIN, а для ознакомления с информацию обо всех магазинах — конструкцию RIGHT JOIN.

Поиск зависших строк, или строк, не имеющих соответствия

Фактически та основная особенность внешних соединений, что полученные с их помощью результаты охватывают все строки одной из таблиц, участвующих в соединении, может использоваться для поиска не имеющих соответствия строк даже в такой таблице, столбцы которой не содержат неопределенных значений (NULL-значений). Что под этим подразумевается? Рассмотрим один пример.

Изменим формулировку рассматривавшегося ранее вопроса, касающегося скидки. Предположим, что требуется определить названия всех магазинов, к которым не относится ни одна строка с данными о скидках. Предлагаем читателю попытаться самому составить необходимый для этого запрос, опираясь на те сведения, которые были изложены к этому моменту. Фактически самый последний используемый в данной главе запрос является почти на 90% подходящим для этой цели. поэтому кратко проанализируем его именно под этим углом: в нем используется внешнее соединение, возвращающее NULL-значения в столбцах с информацией о скидках для тех строк, для которых не найдено соответствие. А в данном случае требуется получить в основном тот же результирующий набор, что и в последнем запросе, не считая того, что должны быть исключены строки, в которых имеется информация о скидках, и оставлены только названия магазинов. Для этого достаточно просто откорректировать в операторе список выборки и ввести конструкцию WHERE. Кроме того, пользователю будет удобнее рассматривать эти данные, если вместо имени поля stor_name будет выведено более понятная строка "Store Name" (Название магазина):

```
SELECT s.stor_name AS "Store Name"
FROM discounts d
RIGHT OUTER JOIN stores s
           ON d.stor_id = s.stor_id
WHERE d.stor_id IS NULL
```

Как и следовало ожидать, получены точно те же названия магазинов, которым в результатах предыдущего запроса соответствовали столбцы с информацией о скидках, заполненные NULL-значениями:

```
Store Name
--------------------------------------------------
Eric the Read Books
Barnum's
News & Brews
Doc-U-Mat: Quality Laundry and Books
Fricative Bookshop

(5 row(s) affected)
```

К этому времени остается неясным только один нюанс, поэтому мы рассмотрим его заранее, не дожидаясь вопросов, чтобы можно было полностью понять, почему описанный здесь метод всегда остается применимым. Указанный нюанс касается того, что до сих пор речь шла о таблицах, в которых заполнены все поля, но на практике встречаются также таблицы, содержащие NULL-значения, и необходимо знать, будет ли описанный здесь подход применяться так же успешно. Дело в том, что в данном запросе в условие конструкции `WHERE` включено имя того же столбца, который вошел в условие соединения. Ведь если соединение выполняется с учетом значений столбцов `stor_id` в обеих таблицах, то могут возникать только три описанных ниже случая.

- Если в поле `stores.stor_id` определенной строки имеется значение, отличное от `NULL`, то согласно операции `ON` конструкции `JOIN` при наличии в таблице `discounts` соответствующей строки поле `discounts.stor_id` также должно иметь значение, равное значению поля `stores.stor_id` (поскольку эта операция имеет вид `ON d.stor_id = s.stor_id`).
- Если в поле `stores.stor_id` определенной строки имеется значение, отличное от `NULL`, то согласно операции `ON` конструкции `JOIN` при отсутствии в таблице `discounts` соответствующей строки значение поля `discounts.stor_id` должно быть возвращено как `NULL`.
- Если оказалось, что поле `stores.stor_id` имеет NULL-значение, а поле `discounts.stor_id` также имеет NULL-значение, то соединение не происходит и вместо значения `discounts.stor_id` возвращается NULL-значение, поскольку соответствие между строками отсутствует.

Итак, при наличии в сравниваемых строках обеих таблиц NULL-значений соединение этих строк не выполняется. Это связано с тем, что, как уже было сказано в отношении сравнения NULL-значений, одно NULL-значение не равно другому. Об этом всегда следует помнить, формулируя текст операторов SQL. Чаще всего вопрос о том, почему данный код не работает, относится к той ситуации, когда предпринимается попытка применить операцию сравнения на равенство к NULL-значениям, и в данном случае ответ найти несложно — попытка сравнить на равенство NULL-значения оканчивается неудачей просто потому, что два NULL-значения не равны друг другу. Чтобы проверить на практике это утверждение, достаточно вызвать на выполнение следующий простой код:

```
IF (NULL=NULL)
  PRINT 'It Does'
ELSE
  PRINT 'It Doesn''t'
```

После вызова на выполнение этого кода СУБД SQL Server сообщает, что результаты операции сравнения одного NULL-значения с другим являются отрицательными, так как на устройстве вывода появляется строка "It Doesn't" (Не равны).

Но фактически получение таких результатов было предусмотрено окончательно лишь в версии SQL Server 7.0. Следует учитывать, что если программа SQL Server 6.5 эксплуатируется в режиме совместимости (в наши дни такая ситуация должна быть достаточно редкой, поскольку режим совместимости относится к давно прошедшим временам, но из всякого правила бывают исключения) или если параметру ANSI_NULLS присвоено значение OFF (с помощью опций сервера или с применением оператора SET), то будет получен другой ответ (сервер будет вырабатывать положительный результат при сравнении двух NULL-значений на равенство). Тем не менее теперь такой режим, в котором сравнение NULL-значений на равенство приводит к получению положительного результата, считается нестандартным. Этот режим рассматривается как нарушение стандарта ANSI и больше не является совместимым с конфигурацией СУБД SQL Server, предусмотренной по умолчанию. (О том, что NULL-значения не равны друг другу, сказано даже в оперативной документации Books Online.)

Воспользуемся рассматриваемым способом, позволяющим выявлять строки, не имеющие несоответствия, для поиска некоторых строк, не вошедших в состав результатов одного из описанных выше операторов с конструкцией INNER JOIN. Напомним, что для выборки данных из базы данных Northwind применялись в том числе следующие два запроса:

```
SELECT DISTINCT c.CustomerID, c.CompanyName
FROM Customers c
INNER JOIN Orders o
      ON c.CustomerID = o.CustomerID
```

и

```
SELECT COUNT(*) AS "No. Of Records" FROM Customers
```

Первый из этих запросов относится к тем примерам, с помощью которых в данной главе рассматривались свойства внутреннего соединения INNER JOIN. А второй запрос позволил выяснить, что в результаты первого запроса не попали (по определению) некоторые строки. Теперь с использованием внешнего соединения определим, какие строки были исключены.

Итак, запрос с конструкцией SELECT COUNT(*) позволил определить, что в результаты первого запроса не вошли некоторые строки из таблицы Customers. (В эти результаты могли также не войти некоторые строки из таблицы Orders, но в данный момент нас это не интересует.) На основании этого можно сделать вывод, что в таблице Customers имеются строки, не находящие соответствия среди строк таблицы Orders. При описании этой темы было принято предположение, что запрос составлен в ответ на требование пользователя получить информацию обо всех заказчиках, разместивших в компании свои заказы, но очень часто приходится искать ответ на прямо противоположный вопрос: "Какие заказчики не разместили в компании свои заказы?" Этот вопрос по сути равнозначен такому вопросу: "Какие строки в таблице Customers не имеют соответствующих строк в таблице Orders?" Решением этой задачи становится запрос с такой же структурой, что и в запросе для поиска магазинов, не предоставляющих скидки:

```
USE Northwind
SELECT c.CustomerID, CompanyName
FROM Customers c
LEFT OUTER JOIN Orders o
          ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL
```

Итак, с помощью одного из предыдущих запросов мы сумели без особых сложностей узнать количество заказчиков, которые не разместили свои заказы, а последний запрос дал возможность определить названия компаний этих заказчиков (и вскоре, возможно, с ними свяжется коммерческий отдел...):

```
CustomerID       CompanyName
-----------      ----------------------------------------------
PARIS            Paris specialites
FISSA            FISSA Fabrica Inter. Salchichas S.A.

(2 row(s) affected)
```

Следует отметить, что результат применения конструкций LEFT JOIN и RIGHT JOIN остается одинаковым, при условии, что для размещения отдельных таблиц или групп таблиц соответствующая часть конструкции, находящаяся слева или справа от конструкции JOIN, выбирается правильно. Например, предыдущий запрос можно было бы с одинаковым успехом составить с использованием конструкции RIGHT JOIN, при условии, что имена таблиц Customers и Orders по отношению к ключевому слову JOIN поменялись бы местами. Например, следующий оператор приводит к получению точно таких же результатов, как и предыдущий:

```
SELECT c.CustomerID, CompanyName
FROM Orders o
RIGHT OUTER JOIN Customers c
          ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL
```

После того как речь пойдет о еще более сложных запросах, будет описан немного более удобный и широко применяемый способ поиска таких строк, имеющихся в одной таблице, для которых отсутствуют соответствующие строки в другой таблице. Тем не менее следует заранее отметить, что обычно соединения позволяют выполнять указанную задачу с наибольшей производительностью. Из этого правила есть исключения, о которых будет сказано при описании соответствующей темы, но, вообще говоря, если имеется несколько способов сравнения таблиц по указанному принципу, то соединения являются наилучшими из них.

Применение более сложных внешних соединений

Переходим к изучению второй из тем, указанных в начале данной главы, а также к рассмотрению практических примеров. В данном разделе речь пойдет исключительно о том, как конструкции OUTER JOIN используются с другими разновидностями конструкции JOIN (независимо от того, каковыми являются эти разновидности).

Понимание того, какие таблицы должны рассматриваться как находящиеся слева или справа от ключевого слова JOIN, становится еще более важным, когда внешнее

соединение применяется в сочетании с другими соединениями. Необходимо учитывать, что с точки зрения включения или исключения из результатов запроса конкретных строк все таблицы, находящиеся "слева" (или перед) от рассматриваемого ключевого слова JOIN, о котором идет речь, должны рассматриваться так, как если бы это была одна таблица. То же утверждение является справедливым по отношению к таблицам, находящимся "справа" (или после) ключевого слова JOIN. Если это не учитывается, то возникает такая распространенная ошибка, как применение в начальной части запроса левого внешнего соединения, а затем выполнения внутреннего соединения в последней части запроса. Ко времени выполнения внутреннего соединения осуществление внешнего соединения приведет к тому, что в результат будут включены все строки одной из таблиц, притом что дальнейшее использование внутреннего соединения все равно может потребовать исключения части результатов! По мнению автора, для большинства разработчиков подобный ход выполнения операций соединения на первых порах кажется весьма труднодоступным для понимания (это касалось и меня самого), поэтому в настоящем разделе суть сказанного будет проиллюстрирована на примерах. Тем не менее ни в одной из баз данных, поставляемых вместе с СУБД SQL Server, нет таких таблиц, которые позволили бы составить удачный сценарий для демонстрации подобных сложных операторов соединения, поэтому вначале необходимо создать отдельную базу данных и ввести в нее данные, которые будут применяться в дальнейших примерах.

Образцовая база данных, называемая Chapter4DB, с помощью которой можно самому проследить за выполнением следующих примеров, может быть создана путем вызова на выполнение сценария Chapter4DB.sql из загруженного с Web-узла автора исходного кода.

Выполнение дальнейших действий будет осуществляться по такому принципу: мы будем шаг за шагом усложнять запрос и следить за тем, что происходит. В конечном итоге должен быть создан запрос, который возвращает имя и адрес поставщика. Количество данных в образцовой базе данных невелико, поэтому начнем с того, что определим все варианты выбора центральной позиции в запросе — имени поставщика. При этом с самого начала (заблаговременно) будем вводить псевдонимы, поскольку желательно, чтобы псевдонимами были обозначены все таблицы в окончательном варианте запроса:

```
SELECT v.VendorName
FROM Vendors v
```

Выполнение этого запроса приводит к получению всего лишь трех строк:

```
VendorName
----------------------------------------
Don's Database Design Shop
Dave's Data
The SQL Sequel

(3 row(s) affected)
```

В состав данных результатов входят имена всех поставщиков, информация о которых имеется к этому времени в базе данных. Теперь получим также информацию об адресах поставщиков. Но при этом возникают две проблемы. Прежде всего необходимо, чтобы запрос возвращал данные всех поставщиков, независимо от того, есть ли о них еще какая-либо информация, поэтому необходимо воспользоваться внешним соединением. Далее, поставщик может иметь больше одного адреса, а по одному адресу могут находиться несколько поставщиков, поэтому в проекте базы данных

было предусмотрено использование связующей таблицы. Это означает, что отсутствуют какие-либо столбцы, позволяющие непосредственно связывать таблицы Vendors и Address; вместо этого необходимо предусмотреть соединение обеих этих таблиц со связующей таблицей, называемой VendorAddress. Начнем с той части соединения, которая по праву должна считаться первой:

```
SELECT v.VendorName
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
          ON v.VendorID = va.VendorID
```

Безусловно, в самой таблице VendorAddress информация об адресах отсутствует, поэтому в список выборки не включены какие-либо столбцы из этой таблицы. Единственное практическое назначение таблицы VendorAddress состоит в том, что она должна быть соединительной точкой связи "многие ко многим" (любой поставщик может иметь много адресов, а также, как уже было сказано, по одному и тому же адресу могут находиться несколько поставщиков). Как и можно было предположить, выполнение этого оператора приводит к получению тех же результатов, как и перед этим:

```
VendorName
--------------------------------------
Don's Database Design Shop
Dave's Data
The SQL Sequel

(3 row(s) affected)
```

На время отвлечемся от данного конкретного запроса, чтобы ознакомиться с той таблицей, с которой было только что сформировано соединение. Попытаемся выполнить выборку всех данных из таблицы VendorAddress:

```
SELECT *
FROM VendorAddress
```

В результате выполнения этого запроса происходит возврат только двух записей:

```
VendorID    AddressID
--------    -------------
1           1
2           3

(2 row(s) affected)
```

Таким образом, подтверждается, что приведенный выше оператор внешнего соединения был выполнен успешно. А поскольку в таблице VendorAddress имеются только две строки, тогда как происходит возврат информации о трех поставщиках, то можно сделать вывод, что выполняется возврат из таблицы Vendors по крайней мере одной строки, не имеющей согласующейся с ней строки в таблице VendorAddress. Продолжая рассматривать данный пример, просто убедимся в этом, введя на время еще один столбец в запрос, касающийся поставщиков:

```
SELECT v.VendorName, va.VendorID
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
           ON v.VendorID = va.VendorID
```

В соответствии с ожиданиями обнаруживается, что в столбце VendorID из таблицы VendorAddress имеется NULL-значение:

```
VendorName                VendorID
------------------------  --------------
Don's Database Design Shop  1
Dave's Data               2
The SQL Sequel            NULL

(3 row(s) affected)
```

Если бы использовалось внутреннее соединение или правое внешнее соединение, то имя поставщика "The SQL Sequel" не было бы возвращено. Возврат данных обо всех поставщиках в результатах запроса был гарантирован благодаря использованию левого внешнего соединения.

Теперь, убедившись в правильности выбранного подхода, возвратимся к первоначальному запросу и введем второе ключевое слово JOIN для получения фактической информации об адресе. На данный момент мы можем не задумываться над тем, получены ли все адреса, поэтому не требуется какая-либо специальная версия оператора соединения; по крайней мере, на первый взгляд не складывается впечатление, что в этом есть необходимость...

```
SELECT v.VendorName, a.Address
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
          ON v.VendorID = va.VendorID
JOIN Address a
  ON va.AddressID = a.AddressID
```

Как и предполагалось, информация об адресах получена, но возникла определенная проблема:

```
VendorName                Address
------------------------  ---------------
Don's Database Design Shop  1234 Anywhere
Dave's Data               567 Main St.

(2 row(s) affected)
```

Дело в том, что была каким-то образом потеряна информация об одном из поставщиков. Причина этого состоит в том, что СУБД SQL Server выполняет конструкции, из которых состоит запрос, в том порядке, в каком они заданы. Вначале было задано ключевое слово OUTER JOIN между именами таблиц Vendors и VendorAddress. СУБД SQL Server выполнила то, что требовалось в этой части запроса, — возвратила информацию обо всех поставщиках. А проблема возникла в связи с применением следующей части оператора. Мы получили результирующий набор, включающий информацию обо всех поставщиках, а затем использовали его как одну из частей (левую), к которой применяется операция внутреннего соединения. Тем не менее операция INNER JOIN действует на обе стороны соединения как исключающая все строки, не находящие взаимного соответствия, поэтому в окончательные результаты включаются только те строки из результирующего набора, полученные в первом соединении, которые согласуются со строками, образующими правую часть второго соединения. А поскольку со строками таблицы Address согласуются только две строки промежу-

точного результирующего набора, то в окончательном результирующем наборе присутствуют лишь две строки. Данную проблему можно устранить одним из двух описанных ниже способов.

- Предусмотреть использование еще одной операции внешнего соединения.
- Изменить последовательность выполнения внешнего и внутреннего соединений.

Проверим оба эти способа. Вначале введем еще одно внешнее соединение:

```
SELECT v.VendorName, a.Address
FROM Vendors v
LEFT OUTER JOIN VendorAddress va
          ON v.VendorID = va.VendorID
LEFT OUTER JOIN Address a
          ON va.AddressID = a.AddressID
```

Теперь действительно получены ожидаемые результаты:

```
VendorName                  Address
--------------------------  ------------
Don's Database Design Shop  1234 Anywhere
Dave's Data                 567 Main St.
The SQL Sequel              NULL

(3 row(s) affected)
```

А на данном этапе внесем гораздо более значительные изменения, связанные с изменением порядка расположения внешнего и внутреннего соединений в первоначальном запросе:

```
SELECT v.VendorName, a.Address
FROM VendorAddress va
JOIN Address a
  ON va.AddressID = a.AddressID
RIGHT OUTER JOIN Vendors v
           ON v.VendorID = va.VendorID
```

И этот вариант приводит к получению желаемого результата:

```
VendorName                  Address
--------------------------  -------------
Don's Database Design Shop  1234 Anywhere
Dave's Data                 567 Main St.
The SQL Sequel              NULL

(3 row(s) affected)
```

Поскольку оказалось, что оба способа составления рассматриваемого запроса являются приемлемыми, возникает резонный вопрос: "Какой из них является наилучшим?" При использовании языка SQL довольно часто обнаруживаются ситуации, в которых можно найти несколько разных способов выполнения одного и того же запроса, притом что ни один из них не имеет существенного преимущества над другими, но в данной ситуации дело обстоит иначе.

В рассматриваемом случае определенно следует порекомендовать использовать второе из возможных решений.

Эмпирическое правило состоит в том, что вначале желательно применить все необходимые операции внутреннего соединения, после чего действительно обнаруживается, что количество требуемых операций внутреннего соединения свелось к минимуму, а количество ошибок в данных уменьшилось.

Причина этого связана с тем, что второй способ позволяет максимально ускорить обработку данных. Если операции внешнего соединения вводятся не для обработки текущей таблицы, информацию которой необходимо включить в результаты запроса, а в связи с тем, что приходится дополнительно обработать результаты текущего соединения, то увеличивается вероятность появления в окончательных результатах тех строк, которые вы не намеревались туда включать, или возникновения какого-то рода ошибки в общей логике формирования оператора. Во втором из рассматриваемых решений такая опасность устранена благодаря использованию внешнего соединения только там, где оно было необходимо, и лишь один раз. В ходе разработки практически используемого оператора не всегда удается так радикально изменить последовательность выполнения операций внешнего и внутреннего соединения, но о такой возможности никогда нельзя забывать.

Следует еще раз подчеркнуть, что ошибки при выборе последовательности выполнения операций соединения встречаются слишком часто. Это – одна из тех областей, в которых знания, приобретенные разработчиками, лишь внешне кажутся достаточными. Автор снова и снова сталкивался с просьбами взглянуть на запрос, над которым другие разработчики провели целые часы, проверяя каждый раздел, причем в половине случаев приходилось слышать, не сталкивался ли я прежде с подобной "программной ошибкой" в СУБД SQL Server. Тем не менее в каждом из таких случаев ошибка обнаруживалась не в СУБД SQL Server, а в коде, составленном разработчиком. Автор надеется, что изучение данного раздела позволяет понять следующее: если полученные результаты не соответствуют ожиданиям, то необходимо прежде всего проверить, нет ли ошибок в выборе последовательности расположения операций соединения.

Просмотр содержимого таблиц, находящихся с обеих сторон от операции соединения, с помощью конструкции FULL JOIN

Как и многие конструкции в языке SQL, конструкция `FULL JOIN` (применяемая также в форме `FULL OUTER JOIN`) по существу выполняет именно то действие, о котором говорит ее название, — эта конструкция согласует данные в таблицах, имена которых находятся по обе стороны от ключевого слова `JOIN`, и вводит в окончательные результаты все строки, независимо от того, с какой стороны соединения они определены.

Конструкции `FULL JOIN` относятся к числу тех языковых средств, которые вызывают восхищение во время их изучения, но в дальнейшем почти не применяются. Чаще можно встретить поистине честного политического деятеля, чем используемую на практике конструкцию `FULL JOIN`! Основное назначение этой конструкции состоит в том, что она позволяет увидеть полную связь между данными в таком виде, в котором не дается преимущество ни левой, ни правой стороне. Эта конструкция

применяется, если есть необходимость ознакомиться с каждой строкой всех таблиц, находящихся по обе стороны от ключевого слова JOIN, без каких-либо исключений.

По-видимому, если одно и то же соединение может быть применено и в форме левого, и в форме правого соединения, то лучше всего использовать полное соединение, имеющее форму конструкции FULL JOIN. Эта конструкция не только дает возможность получить все согласующиеся строки с учетом того поля (полей), на котором основано соединение, но и те строки, которые имеются только в таблицах, находящихся на левой стороне, притом что столбцы, относящиеся к правой стороне, заполняются NULL-значениями. Наконец, та же операция возвращает все строки, имеющиеся только в таблицах, заданных с правой стороны, а вместо значений полей таблиц, относящихся к левой стороне, подставляются NULL-значения. Следует отметить, что слово "наконец" в предыдущем предложении не означает, что обработка в запросе таблиц, находящихся справа от ключевого слова JOIN, происходит в последнюю очередь. Как показывает анализ последовательности расположения строк в результирующем наборе (остающейся такой же, как и после формирования результирующего набора, если не используется конструкция ORDER BY), порядок обработки таблиц полностью зависит от решения по выбору наименее дорогостоящего способа выборки строк, принятого СУБД SQL Server.

Практическое занятие

Выполнение операции полного соединения с помощью конструкции FULL JOIN

Вначале еще раз рассмотрим последний запрос из раздела, посвященного внешним соединениям:

```
SELECT v.VendorName, a.Address
FROM VendorAddress va
JOIN Address a
   ON va.AddressID = a.AddressID
RIGHT OUTER JOIN Vendors v
            ON v.VendorID = va.VendorID
```

Теперь наша задача состоит в том, чтобы последовательно дополнять отдельные части исходного оператора и вводить некоторые поля в список выборки, проверяя, что при этом происходит. Вначале выполним соединение двух первых таблиц с использованием конструкции FULL JOIN:

```
SELECT a.Address, va.AddressID
FROM VendorAddress va
FULL JOIN Address a
      ON va.AddressID = a.AddressID
```

Как оказалось, первый оператор FULL JOIN, рассматриваемый в этом разделе, не позволяет получить больше информации, чем оператор JOIN RIGHT:

```
Address              AddressID
---------------      -------------
1234 Anywhere        1
567 Main St.         3
999 1st St.          NULL
1212 Smith Ave       NULL
364 Westin           NULL

(5 row(s) affected)
```

Тем не менее с помощью операции полного соединения можно достичь гораздо большего! Введем еще одну конструкцию JOIN:

```
SELECT a.Address, va.AddressID, v.VendorID, v.VendorName
FROM VendorAddress va
FULL JOIN Address a
      ON va.AddressID = a.AddressID
FULL JOIN Vendors v
      ON va.VendorID = v.VendorID
```

Теперь в нашем распоряжении имеется оператор, позволяющий получить из рассматриваемых таблиц всю имеющуюся в них информацию:

```
Address          AddressID      VendorID       VendorName
---------------  -----------    -----------    -------------------------
1234 Anywhere    1              1              Don's Database Design Shop
567 Main St.     3              2              Dave's Data
999 1st St.      NULL           NULL           NULL
1212 Smith Ave   NULL           NULL           NULL
364 Westin       NULL           NULL           NULL
NULL             NULL           3              The SQL Sequel

(6 row(s) affected)
```

Описание полученных результатов

Вполне очевидно, что в первую очередь были сформированы те же две строки, которые могли быть получены с помощью конструкции INNER JOIN. За ними следуют три строки из таблицы Address, которые не были согласованы с какими-либо другими строками той или другой из оставшихся таблиц. Наконец, получена одна строка из таблицы Vendors, с которой не согласуются какие-либо другие строки (но возможности полного соединения на этом не исчерпываются).

Еще раз отметим, что конструкция FULL JOIN должна использоваться, если требуется получить все строки из таблиц, находящихся по обе стороны от ключевого слова JOIN; если это возможно, строки будут согласованы, но будут также включены, даже если согласование не произошло.

Конструкция CROSS JOIN

Операторы с конструкциями CROSS JOIN обладают действительно необычными особенностями. Соединения CROSS JOIN отличаются от соединений других типов тем, что в них отсутствуют операции ON, а также тем, что в них происходит соединение каждой строки таблиц, находящихся с одной стороны от ключевого слова JOIN, с каждой строкой таблиц, находящихся с другой стороны от ключевого слова JOIN. Короче говоря, в конечном итоге формируется декартово произведение всех строк, заданных по обе стороны от ключевого слова JOIN. Операторы с конструкцией CROSS JOIN имеют такой же синтаксис, как и любые другие операторы JOIN, за исключением того, что в них используется ключевое слово CROSS (вместо INNER, OUTER или FULL), а операция ON отсутствует. Ниже приведен краткий пример.

```
SELECT v.VendorName, a.Address
FROM Vendors v
CROSS JOIN Address a
```

Теперь рассмотрим, какова структура обрабатываемых таблиц: в таблице Vendors имеются три строки, а в таблице Address — пять строк. Если каждая строка таблицы Vendors будет согласована с каждой строкой таблицы Address, то в конечном итоге результат выполнения операции CROSS JOIN будет состоять из 3 × 5 = 15 строк:

```
VendorName                    Address
----------------------------  --------------------
Don's Database Design Shop    1234 Anywhere
Don's Database Design Shop    567 Main St.
Don's Database Design Shop    999 1st St.
Don's Database Design Shop    1212 Smith Ave
Don's Database Design Shop    364 Westin
Dave's Data                   1234 Anywhere
Dave's Data                   567 Main St.
Dave's Data                   999 1st St.
Dave's Data                   1212 Smith Ave
Dave's Data                   364 Westin
The SQL Sequel                1234 Anywhere
The SQL Sequel                567 Main St.
The SQL Sequel                999 1st St.
The SQL Sequel                1212 Smith Ave
The SQL Sequel                364 Westin

(15 row(s) affected)
```

В данном случае действительно получены те результаты, которых следовало ожидать.

Автор читает лекции по языку SQL и каждый раз, рассказывая о конструкции CROSS JOIN, слышит один и тот же вопрос: "Для чего может быть предназначена такая операция?" На этот вопрос можно дать такой ответ, что декартово произведение имеет широкую область применения в науке; в частности, наличие этой конструкции оправдано, поскольку имеется целый ряд математических функций высокого уровня, в которых применяются декартовы произведения. А на практике вполне может использоваться такой способ обработки данных, в котором в табличные структуры считывается большое количество образцов данных, а затем выполняется операция перекрестного соединения CROSS JOIN для создания декартова произведения этих образцов. Тем не менее операторы CROSS JOIN гораздо чаще предназначены для создания испытательных данных.

Чаще всего формирование базы данных осуществляется с учетом того, что эта база данных войдет в состав более крупномасштабной системы, требующей существенной проверки. А при тестировании систем большого масштаба снова и снова возникает проблема, связанная с высокой трудоемкостью создания больших объемов данных, применяемых при испытаниях. Использование операции CROSS JOIN открывает такую возможность, что могут быть созданы две или несколько таблиц с количеством строк испытательных данных, намного меньшим по сравнению с требуемым. После этого к таким промежуточным таблицам можно применить операторы CROSS JOIN для создания гораздо более крупных наборов испытательных данных. Наглядный пример применения такого подхода представляет собой последний запрос. Он по-

казывает, что если требуется выполнить проверку приложения по данным, в которые входят адреса и имена поставщиков, то даже простой запрос позволяет получить 15 строк из 8. Безусловно, эти цифры могут стать намного более существенными. Например, если бы была создана таблица имен с 50 именами и таблица фамилий с 250 фамилиями, то применение оператора `CROSS JOIN` для соединения этих таблиц позволило бы создать таблицу с 12 500 уникальными комбинациями имен и фамилий. Это означает, что достаточно ввести вручную 300 строковых значений с именами и фамилиями, после чего сразу же получить набор испытательных данных с 12 500 именами и фамилиями!

Альтернативный синтаксис операторов соединений

В настоящем разделе рассматривается синтаксис, не соответствующий современному стандарту, но все еще рассматриваемый многими как "нормальный" способ составления кода операторов соединения. До появления версии SQL Server 6.5 приведенный в этом разделе альтернативный синтаксис был единственно возможным синтаксисом операторов соединения СУБД SQL Server, а тот способ оформления операторов соединения, который применяется в настоящее время и который принято называть "стандартным", даже еще не был предусмотрен.

До сих пор во всех операторах SQL, рассматриваемых в данной книге, использовался синтаксис, который определен стандартом ANSI. Автор настоятельно рекомендует опираться именно на стандарт ANSI, поскольку он обеспечивает гораздо лучшую переносимость между системами, а также позволяет создавать операторы, намного более удобные для чтения. Но следует отметить, что и указанный устаревший синтаксис в настоящее время фактически достаточно полно поддерживается на разных платформах.

Основная причина, по которой в данной главе рассматривается устаревший синтаксис, состоит в том, что любому разработчику рано или поздно придется столкнуться с ним в унаследованном коде, и в этом нет ни малейшего сомнения. Автор не может допустить, чтобы читатели настоящей книги когда-либо недоуменно рассматривали подобный код, говоря про себя: "Что бы это все значило?"

Несмотря на сказанное, еще раз повторим настоятельную рекомендацию, что при любой возможности следует использовать синтаксис ANSI. Еще раз отметим, что операторы, оформленные с использованием современного синтаксиса, становятся гораздо более удобными для чтения, а представители компании Microsoft дали понять, что поддержка устаревшего синтаксиса не может продолжаться до бесконечности. Безусловно, если учесть, насколько велик объем все еще эксплуатируемого унаследованного кода, трудно поверить, что вскоре настанет такое время, когда компания Microsoft запретит использование устаревшего синтаксиса, но никто не может гарантировать обратное.

По-видимому, самая важная причина отказа от устаревшего синтаксиса состоит в том, что синтаксис ANSI фактически предоставляет большие функциональные возможности. К тому же устаревший синтаксис действительно допускал создание запросов, имеющих неоднозначное толкование. Иными словами, иногда было возможно интерпретировать тексты запросов по-разному. А новый синтаксис устраняет эту проблему.

Напомним, что выше в данной главе приведено сравнение конструкции JOIN с конструкцией WHERE. Это сравнение применялось еще с одной целью. Дело в том, что в устаревшем синтаксисе все условия выполнения соединений выражаются в конструкции WHERE.

Устаревший синтаксис поддерживает все те соединения, которые были описаны в данной главе на основе синтаксиса ANSI, за исключением операторов FULL JOIN. Поэтому даже тем разработчикам, которые упорно придерживаются устаревшего синтаксиса, для выполнения полного соединения приходится прибегать к использованию синтаксиса ANSI.

Синтаксис, альтернативный по отношению к синтаксису оператора INNER JOIN

Еще раз вернемся к первому оператору INNER JOIN, который рассматривался в данной главе:

```
SELECT *
FROM Products
INNER JOIN Suppliers
      ON Products.SupplierID = Suppliers.SupplierID
```

Выполнение этого оператора приводит к получению 77 строк (опять-таки при условии, что база данных Northwind осталась той же, какой была сразу после ее развертывания в составе дистрибутива СУБД SQL Server). Но вместо использования конструкции JOIN, предусмотренной стандартом ANSI, переформулируем этот запрос на основе синтаксиса соединения, предусматривающего применение конструкции WHERE. Такая задача дсйствительно является весьма несложной — достаточно исключить слова INNER JOIN и ввести запятую, а затем заменить операцию ON конструкцией WHERE:

```
SELECT *
FROM Products, Suppliers
WHERE Products.SupplierID = Suppliers.SupplierID
```

На этом внесение изменений заканчивается, а после вызова полученного оператора на выполнение формируются те же 77 строк, которые были получены с применением другого синтаксиса.

Рассматриваемый в этом разделе синтаксис в настоящее время поддерживается практически всеми основными системами, в которых предусмотрено использование языка SQL (Oracle, DB2, MySQL и т.д.).

Синтаксис, альтернативный по отношению к синтаксису OUTER JOIN

Синтаксис операторов, альтернативный по отношению к синтаксису с конструкцией OUTER JOIN, во многом напоминает тот, который применяется вместо синтаксиса INNER JOIN, за исключением того, что он требует использования некоторых особых знаков операций, специально предусмотренных для этой задачи, поскольку альтернативный синтаксис не поддерживает ключевые слова LEFT и RIGHT (а также в связи с этим ключевые слова OUTER и JOIN). Указанные знаки операций приведены в табл. 4.12.

Таблица 4.12. Альтернативный и стандартный синтаксис операторов внешнего соединения

Альтернативный синтаксис	Синтаксис по стандарту ANSI
*=	LEFT JOIN
=*	RIGHT JOIN

Рассмотрим в качестве примера первый оператор с конструкцией OUTER JOIN, который был описан в данной главе. В этом операторе, приведенном ниже, использовалась база данных pubs.

```
SELECT discounttype, discount, s.stor_name
FROM discounts d
LEFT OUTER JOIN stores s
          ON d.stor_id = s.stor_id
```

В этом случае также достаточно исключить слова LEFT OUTER JOIN и заменить операцию ON конструкцией WHERE:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d, stores s
WHERE d.stor_id *= s.stor_id
```

Как и следовало ожидать, и в данном случае формируются такие же результаты, как и прежде:

```
discounttype          discount      stor_name
------------------    ---------     --------------
Initial Customer      10.50         NULL
Volume Discount       6.70          NULL
Customer Discount     5.00          Bookbeat

(3 row(s) affected)
```

Оператор с конструкцией RIGHT JOIN принимает практически такой же вид:

```
SELECT discounttype, discount, s.stor_name
FROM discounts d, stores s
WHERE d.stor_id =* s.stor_id
```

И в этом случае выполнение оператора приводит к получению того же количества строк (шести), как и при использовании оператора с синтаксисом ANSI.

В последнем примере результаты применения двух операторов, имеющих разный синтаксис, не расходятся, но так было не всегда, поскольку до публикации синтаксического определения внешнего соединения по стандарту ANSI поддержка внешних соединений в разных СУБД действительно характеризовалась отсутствием единообразия. Основные возможности операций внешнего соединения совпадали, но нюансы изменялись от системы к системе и от версии к версии.

Синтаксис, альтернативный по отношению к синтаксису CROSS JOIN

Приведение операторов с конструкцией CROSS JOIN к альтернативному синтаксису является намного более простой задачей по сравнению со всеми другими конструкциями соединений. Чтобы создать с помощью альтернативного синтаксиса оператор,

равнозначный оператору с конструкцией CROSS JOIN, достаточно просто оставить его в неизменном виде. Иными словами, ничего не нужно включать в конструкцию WHERE в форме TableA.ColumnA = TableB.ColumnA.

Итак, в качестве чрезвычайно простого примера рассмотрим первый вариант оператора из раздела, посвященного описанию синтаксиса CROSS JOIN, который находится выше в данной главе. Оператор, соответствующий синтаксису ANSI, выглядит следующим образом:

```
SELECT v.VendorName, a.Address
FROM Vendors v
CROSS JOIN Address a
```

Чтобы преобразовать его в оператор с альтернативным синтаксисом, достаточно удалить ключевое слово CROSS JOIN и ввести запятую:

```
SELECT v.VendorName, a.Address
FROM Vendors v, Address a
```

Как и при выполнении других примеров данного раздела, будут получены те же результаты, как и с помощью операторов с синтаксисом ANSI:

```
VendorName                    Address
----------------------------  --------------------
Don's Database Design Shop    1234 Anywhere
Don's Database Design Shop    567 Main St.
Don's Database Design Shop    999 1st St.
Don's Database Design Shop    1212 Smith Ave
Don's Database Design Shop    364 Westin
Dave's Data                   1234 Anywhere
Dave's Data                   567 Main St.
Dave's Data                   999 1st St.
Dave's Data                   1212 Smith Ave
Dave's Data                   364 Westin
The SQL Sequel                1234 Anywhere
The SQL Sequel                567 Main St.
The SQL Sequel                999 1st St.
The SQL Sequel                1212 Smith Ave
The SQL Sequel                364 Westin

(15 row(s) affected)
```

Итак, теперь описаны конструкции и варианты синтаксиса, применяемые в большинстве систем управления базами данных.

Операция UNION

На этом сравнение нового и альтернативного синтаксиса операторов соединения заканчивается. Перейдем к описанию операции UNION, которая всегда имеет одинаковую форму, независимо от того, какова в остальном синтаксическая форма оператора соединения. UNION — это специальная операция, с помощью которой можно создавать общий результирующий набор, используя данные, полученные с помощью двух или нескольких запросов.

В действительности, в отличие от предыдущих рассматриваемых вариантов, операция UNION не представляет собой соединение, поскольку она в большей степени

напоминает способ добавления данных из одного запроса непосредственно к концу данных, полученных с помощью другого запроса (с функциональной точки зрения выполняемые при этом действия выглядят немного иначе, но именно такая трактовка позволяет легче всего понять, как действует рассматриваемая конструкция). Конструкция JOIN обеспечивает увеличение количества полей в строках (добавление столбцов), а конструкция UNION позволяет увеличить длину столбцов (добавляя больше строк), как показано на рис. 4.1.

Анализируя запросы, содержащие ключевое слово UNION, или формируя такие запросы, достаточно учитывать лишь несколько описанных ниже основных соображений.

- Все запросы, результаты которых объединяются с помощью конструкции UNION, должны иметь одинаковое количество столбцов в списке выборки. Если количество столбцов в списке выборки первого запроса равно трем, то второй запрос (а также все последующие запросы, соединяемые операцией UNION) также должен иметь в списке выборки три столбца. А если в первом списке выборки предусмотрено пять столбцов, то и во втором их должно быть пять. В каждом из последующих запросов всегда должно быть такое же количество столбцов, как в первом.

- Заголовки для столбцов комбинированного результирующего набора берутся только из формулировки первого запроса. Если первый запрос имеет список выборки, который выглядит, допустим, как SELECT Col1, Col2 AS Second, Col3 FROM..., то независимо от имен или псевдонимов столбцов в последующих запросах заголовками столбцов, возвращенных оператором с конструкцией UNION, будут соответственно Col1, Second и Col3.

- Типы данных каждого столбца, указанного в любом запросе, должны быть неявно совместимыми с типами данных столбцов, занимающих такое же относительное положение в списках выборки других запросов. Обратите внимание на то, что речь не идет о наличии в соответствующих столбцах данных одинаковых типов; достаточно лишь того, чтобы используемые данные допускали неявное преобразование типов (таблица преобразований, в которой показаны явные и неявные преобразования, приведена в главе 2). Если второй столбец в первом запросе относится к типу char(20), то вполне допустимо, чтобы второй столбец во втором запросе имел тип varchar(50). Тем не менее за основу берется первый запрос, поэтому при обработке данных из второго результирующего набора все строковые значения, превышающие по длине 20 символов, будут усечены справа.

- В запросах с конструкцией UNION в отличие от тех запросов, в которых не применяется эта конструкция, по умолчанию принято использование опции DISTINCT, а не ALL. Из-за этого на первых порах при освоении запросов такого типа может возникать значительная путаница. Ведь при выполнении других запросов происходит возврат всех строк, независимо от того, являются ли они дубликатами по отношению к другим строкам или нет, но результаты запросов с конструкцией UNION выглядят иначе — если в запросе не используется ключевое слово ALL, то происходит возврат только одной из всех дублирующихся строк.

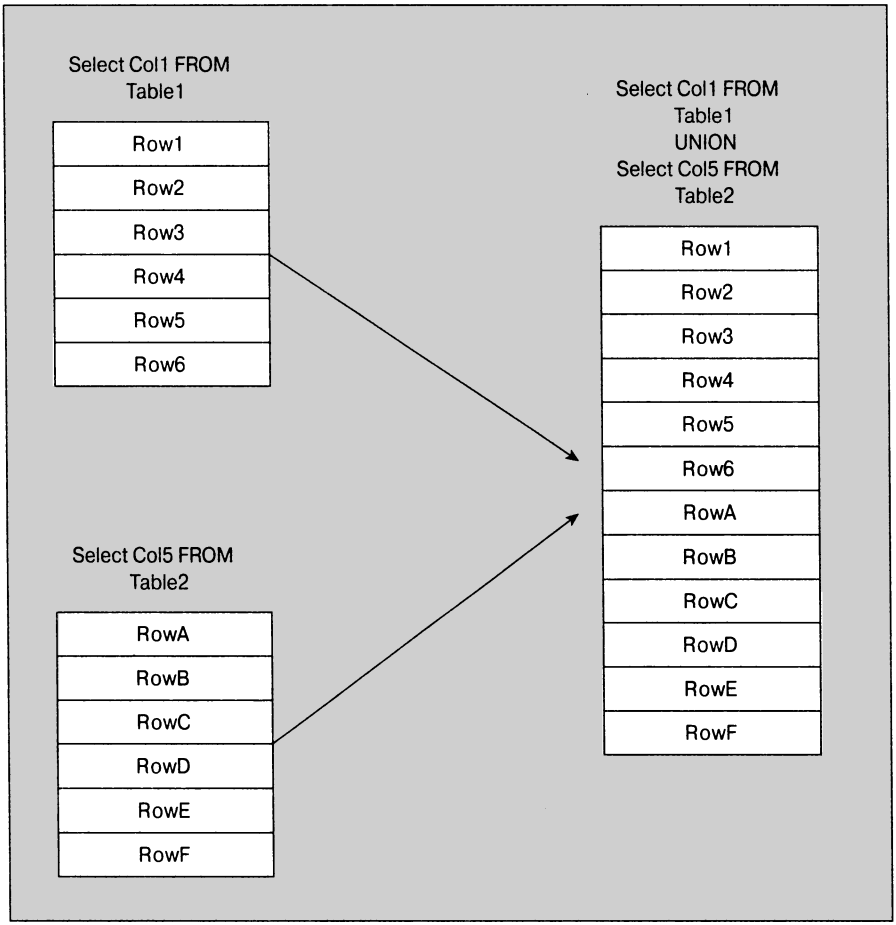


Рис. 4.1. Схематическая иллюстрация процесса формирования результатов объединения

Как обычно, рассмотрим несколько примеров использования запросов с конструкцией `UNION`.

Практическое занятие

Применение запроса с конструкцией `UNION`

Вначале рассмотрим запрос с конструкцией `UNION`, который действительно может найти практическое применение (автор убедился в том, что подобные запросы действительно бывают нужны в реальном мире, хотя и не очень часто). Например, предположим, что приближаются новогодние каникулы и мы хотим отправить открытку и поздравить с Новым годом всех тех, кто имеет отношение к компании Northwind. Для этого необходимо составить список полных адресов, по которым должны быть отправлены открытки, и включить туда служащих, заказчиков и поставщиков компании. Такую задачу можно выполнить, воспользовавшись всего лишь одним запросом, подобным приведенному ниже.

```
USE Northwind
SELECT CompanyName AS Name,
      Address,
      City,
      Region,
      PostalCode,
      Country
FROM Customers
UNION
SELECT CompanyName,
      Address,
      City,
      Region,
      PostalCode,
      Country
FROM Suppliers
UNION
SELECT FirstName + ' ' + LastName,
      Address,
      City,
      Region,
      PostalCode,
      Country
FROM Employees
```

Этот запрос возвращает только один результирующий набор (табл. 4.13), но в нем содержатся данные из всех трех запросов.

Таблица 4.13. Результаты выполнения оператора соединения

Name	Address	City	Region	PostalCode	Country
Alfreds Futterkiste	Obere Str. 57	Berlin	NULL	12209	Germany
Ana Trujillo Emparedados y helados	Avda. de la Constitucion 2222	Mexico D.F.	NULL	5021	Mexico
Andrew Fuller	908 W. Capital Way	Tacoma	WA	98401	USA
...					
...					
...					
Wilman Kala	Keskuskatu 45	Helsinki	NULL	21240	Finland
Wolski Zajazd	ul. Filtrowa 68	Warszawa	NULL	01-012	Poland
Zaanse Snoepfabriek	Verkoop Rijnweg 22	Zaandam	NULL	9999 ZZ	Nether-lands

Итак, в этом списке никто не забыт. Компания `Alfreds` — заказчик, `Andrew Fuller` — служащий, а компания `Zaanse` — поставщик.

Во время отладки этого запроса автор получал довольно несовместимые результаты сортировки строк, полученных при выполнении запроса, поэтому не стоит удивляться, если будет получена последовательность строк, значительно отличающаяся от приведенной в табл. 4.13. Однако важнее то, что должно быть получено примерно 129 строк (количество строк изменяется в зависимости от того, какие изменения были внесены в базу данных `Northwind` *перед этим). Если же требуется, чтобы результаты были возвращены в определенном порядке, то следует применить конструкцию* `ORDER BY`*. В операторах с конструкцией* `UNION` *конструкция* `ORDER BY` *должна входить в состав последнего запроса, на который распространяется операция объединения (*`UNION`*).*

Описание полученных результатов

Вместо трех результирующих наборов получен только один.

СУБД SQL Server выполняет все эти три запроса и по существу добавляет результаты, полученные позже, к результатам, сформированным раньше, создавая один объединенный результирующий набор. Еще раз следует подчеркнуть, что заголовки для возвращаемых столбцов берутся из списка выборки первого запроса.

Переходя ко второму примеру, автор хочет показать, какие действия осуществляются оператором `UNION` применительно к дублирующимся строкам; по существу при обработке дублирующихся строк происходят действия, обратные по отношению к обычному запросу, поскольку по умолчанию в процессе объединения результирующих наборов дубликаты строк отбрасываются. (При описании запросов других типов было принято предположение, что необходимо сохранить все строки в результатах, если не задано ключевое слово `DISTINCT`.) Данный пример вряд ли найдет применение на практике, но он — краткий, может быть легко выполнен, а также позволяет полностью разобраться в том, что происходит.

В данном случае создаются две таблицы, из которых должна осуществляться выборка данных. После этого происходит вставка, по три строки в каждую таблицу, причем одна строка повторяется в обеих таблицах. Если бы в выполняемом запросе было задано ключевое слово `ALL`, то в результат вошли бы все строки (в данном случае шесть). А если запрос выполняется с ключевым словом `DISTINCT`, то количество возвращаемых строк должно составлять только пять (поскольку одна дублирующаяся строка отбрасывается):

```
CREATE TABLE UnionTest1
(
  idcol   int      IDENTITY,
  col2    char(3),
)
CREATE TABLE UnionTest2
(
  idcol   int      IDENTITY,
  col4    char(3),
)
INSERT INTO UnionTest1
VALUES
  ('AAA')
INSERT INTO UnionTest1
VALUES
  ('BBB')
```

```
INSERT INTO UnionTest1
VALUES
   ('CCC')
INSERT INTO UnionTest2
VALUES
   ('CCC')
INSERT INTO UnionTest2
VALUES
   ('DDD')
INSERT INTO UnionTest2
VALUES
   ('EEE')
SELECT col2
FROM UnionTest1
UNION
SELECT col4
FROM UnionTest2
PRINT 'Divider Line--------------------------'
SELECT col2
FROM UnionTest1
UNION ALL
SELECT col4
FROM UnionTest2
DROP TABLE UnionTest1
DROP TABLE UnionTest2
```

Ниже приведена только основная часть полученных результатов (по мере выполнения отдельных операторов происходил возврат сообщений "`one row(s) affected`", но они здесь не показаны, а приведены только данные, позволяющие ознакомиться с интересующими нас результатами запросов).

```
col2
------
DDD
EEE
AAA
BBB
CCC
(5 row(s) affected)
Divider Line--------------------------
col2
-----
AAA
BBB
CCC
CCC
DDD
EEE

(6 row(s) affected)
```

Первый результирующий набор был получен с помощью обычного оператора `UNION` без дополнительных параметров. Вполне очевидно, что в нем удалена одна строка, — хотя строка "`CCC`" была вставлена в обе таблицы, появился только один ее экземпляр, поскольку дублирующиеся строки уничтожаются по умолчанию.

Второй результирующий набор немного изменился. На этот раз использовалась конструкция UNION ALL, а ключевое слово ALL гарантирует возврат всех полученных строк. Именно поэтому внезапно появилась дублирующаяся строка, которая была удалена из результатов предыдущего запроса.

Резюме

В реляционной СУБД данные чаще всего не сосредоточиваются в одной таблице, а распределяются по нескольким таблицам. Конструкции JOIN и UNION позволяют комбинировать данные из нескольких таблиц с применением описанных ниже способов.

- Если требуется исключить из полученных результатов данные, которые не имеют согласующихся полей, то используется конструкция INNER JOIN.
- Если требуется сохранить в составе результатов все данные, поля в которых согласуются, и вместе с тем дополнить их результирующим набором, содержащим полный объем данных, относящихся к одной из сторон соединения, то используется конструкция OUTER JOIN.
- Если требуется сохранить в составе результатов все данные, поля в которых согласуются, и вместе с тем дополнить их результирующим набором, содержащим полный объем данных, относящихся к обеим сторонам соединения, то используется конструкция FULL JOIN.
- Если требуется получить декартово произведение, в состав которого входят записи двух таблиц, то используется конструкция CROSS JOIN. Как правило, декартовы произведения применяются при формировании входных данных для некоторых сложных функций, а также служат для создания испытательных данных.
- Если требуется объединить результаты второго запроса с результатами первого запроса, то используется конструкция UNION.

Синтаксис операторов внутренних и внешних соединений имеет две различные формы. В настоящей главе, кроме стандартного, представлен альтернативный синтаксис, чтобы читатель не испытывал затруднений, сталкиваясь с унаследованным кодом (в котором встречается альтернативный синтаксис). Однако настоятельно рекомендуется использовать новый формат операторов, предусмотренный стандартом ANSI, которому посвящена основная часть настоящей главы, поскольку он является более удобным для чтения, не допускает возникновения неоднозначностей, характерных для устаревшего синтаксиса, а его поддержка в СУБД SQL Server всегда гарантирована.

В следующих нескольких главах будет показано, как создавать собственные таблицы и связывать их друг с другом. По мере изучения этого материала основные принципы формирования соединений таблиц с использованием согласуемых друг с другом столбцов станут еще более очевидными.

Упражнения

4.1. Напишите запрос к базе данных `Northwind`, который возвращает один столбец `SupplierName` и содержит имя поставщика товара `Chai`.

4.2. Напишите запрос с конструкцией `JOIN`, чтобы получить список с указанием каждой территории в базе данных `Northwind`, для обслуживания которой не назначен служащий. (Подсказка. Используйте внешнее соединение.)

5

Создание и модификация таблиц

Студенты, которые изучают, как применяется код T-SQL для создания баз данных, таблиц, ключей и ограничений, часто задают один и тот же вопрос: "Нельзя ли выполнить все эти действия с помощью инструментального средства с графическим интерфейсом пользователя?" Ответ на этот вопрос, безусловно, является положительным. Но за этим вопросом обычно сразу же следует другой: "Так зачем же тратить время на изучение того, что никогда не потребуется в работе?" Ответ является столь же однозначным — на практике время от времени все равно приходится использовать обычный синтаксис. В действительности нет необходимости неизменно разрабатывать код создания и модификации объектов базы данных с нуля, но в большинстве крупных проектов баз данных этот код приходится проверять и редактировать, а это означает, что необходимо разбираться в том, как он работает.

Настоящая глава посвящена изучению синтаксиса операторов, применяемых для создания пользователем собственных таблиц. Кроме того, в ней описано, как воспользоваться программой SQL Management Console для упрощения этой работы (это описание будет приведено после изучения способов самостоятельного выполнения всех необходимых для этого задач).

Прежде чем перейти к глубокому изучению операторов, фактически используемых для создания таблиц и других объектов, мы должны сделать достаточно большое отступление, чтобы рассмотреть, какие соглашения распространяются на структуру полностью уточненных имен объектов, а также в меньшей степени на права использования объектов.

Структура имен объектов в СУБД SQL Server

Во всех запросах, которые рассматривались до сих пор в настоящей книге, использовались простые средства именования. Перед выполнением любых запросов автор предлагал активизировать требуемую базу данных в программе Query Analyzer, благодаря чему обеспечивалось выполнение запросов применительно к тому объекту, для которого они предназначены. В этих случаях необходимость переключения на другую базу данных была связана с тем, что при использовании имен, не имеющих уточнений, СУБД SQL Server, осуществляя попытки идентифицировать и найти объекты, указанные в запросах и других операторах, рассматривает только очень узкую область определения. Таким образом, в рассматриваемых до сих пор примерах было предусмотрено применение только имен таблиц, без какой-либо дополнительной информации, но фактически для именования любых таблиц СУБД SQL Server (а также, в этом контексте, любых других объектов SQL Server) предусмотрено соглашение, допускающее использование четырех уровней именования. Полностью уточненное имя имеет такую структуру:

```
[ServerName.[DatabaseName.[SchemaName.]]]ObjectName
```

При выполнении любой операции над объектом необходимо указать имя этого объекта, но все остальные части уточненного имени, находящиеся слева от имени объекта, `ObjectName`, являются необязательными. И действительно, чаще всего без дополнительных уточнителей вполне можно обойтись, поэтому они не применяются. Тем не менее, прежде чем приступить к созданию объектов, следует полностью разобраться в том, что означает каждая часть уточненного имени. Ниже компоненты структуры уточненного имени рассматриваются справа налево.

Имя схемы (или обозначение принадлежности)

Схемой (schema) называется часть базы данных, принадлежащая определенному пользователю. В большинстве создававшихся ранее баз данных принцип распределения объектов по схемам не использовался, но складывается впечатление, что необходимость реализации этого принципа в дальнейшем будет становиться все более и более насущной. Если в базе данных используются схемы, то для обеспечения доступа к объектам требуется указывать, в какой схеме находятся эти объекты. Вполне допустимо, чтобы в базе данных имелись два объекта с одинаковыми именами, находящиеся в разных схемах. Если пользователь желает получить доступ к объекту, который не находится в схеме этого пользователя, применяемой по умолчанию (которая определяется во время регистрации пользователя), то он должен конкретно указывать имя схемы, `SchemaName`, в которой находится объект. Например, рассмотрим вариант применения схем в базе данных `AdventureWorks` (кстати, это — один из наихудших вариантов организации схем, которые когда-либо встречались автору) и составим запрос на получение списка сотрудников с указанием городов, в которых они проживают:

```
SELECT e.EmployeeID, c.FirstName, c.LastName, City
FROM HumanResources.Employee AS e
JOIN Person.Contact c
 ON e.ContactID = c.ContactID
```

```
JOIN HumanResources.EmployeeAddress AS ea
 ON e.EmployeeID = ea.EmployeeID
JOIN Person.Address AS a
 ON ea.AddressID = a.AddressID
```

В данном примере используются четыре таблицы, распределенные по двум схемам. Если бы оказалось так, что одна из двух рассматриваемых схем (HumanResources или Person) является схемой, применяемой по умолчанию, то можно было бы не задавать имя этой схемы, составляя в запросе имена таблиц, находящихся в схеме. Но в данном случае указаны имена всех схем, поскольку это позволяет избежать любых неприятных сюрпризов.

Настоящий пример иллюстрирует еще одну ситуацию, в которой автор неизменно настаивает, что нужно придерживаться единообразия. Если в процессе работы приходится использовать средства базы данных, предусматривающие распределение объектов по схемам, то автор настоятельно рекомендует придерживаться во всех запросах соглашения об именовании, в соответствии с которым каждое имя должно состоять из двух частей (имя схемы и имя таблицы). Дело в том, что слишком легко могут возникнуть такие предпосылки нарушения в работе, при которых изменится схема, применяемая пользователем по умолчанию, или будет переопределен какой-то другой псевдоним, в результате чего предположения, касающиеся выбора компонентов имен, заданных по умолчанию, станут недействительными. Если до сих пор во всех ваших проектах баз данных не использовался принцип распределения объектов по разным схемам, то имеет смысл оставить все эти проекты в неизменном виде (что способствует также значительному повышению удобства чтения кода), но если в дальнейшем вы перейдете к использованию схем, то придется столкнуться с дополнительными сложностями.

Некоторые вспомогательные сведения о схемах

В стандарте ANSI языка SQL понятие, связанное со структуризацией баз данных, получившее название схемы, было сформулировано уже довольно давно. Следует отметить, что в СУБД SQL Server аналогичная концепция была реализована с самого начала, но для ее именования использовались другие термины (а также, безусловно, даже если бы можно было применить для обозначения этого понятия то же название, оно имело бы немного другое назначение). Поэтому, в частности, в предыдущих версиях SQL Server то, что принято называть "схемой" в SQL Server 2005 и других базах данных, таких как Oracle, обычно именовалось "владельцем".

Подход, предусматривающий применение способа структуризации баз данных с помощью схем, выдержал проверку временем. Безусловно, сама реализация этого способа все еще остается достаточно сложной, но компания Microsoft ввела несколько новых усовершенствований в программное обеспечение SQL Server 2005, благодаря чему решение проблем, связанных с использованием схем, значительно упростилось. Но если в процессе работы необходимо обеспечивать обратную совместимость с предыдущими версиями SQL Server, то следует либо полностью избегать применения новых средств, либо учитывать все нюансы, связанные с использованием устаревших, а это означает, что стремление реализовать в новых проектах понятие "владельца" (предусмотренное в предыдущих версиях) становится источником существенных затруднений.

Разумеется, определенная часть разработчиков всегда с трудом расстается с теми средствами, которые должны быть подвергнуты пересмотру. В качестве примера

можно привести использование понятия "владельца" в версиях, предшествующих SQL Server 2005, но автор, безусловно, не относится к такой категории разработчиков. Важной отличительной особенностью версии SQL Server 2005, о которой всегда следует помнить, является то, что в ней изменилась структура имен в связи с отказом от понятия "владельца", поэтому та часть базы данных, в которой находятся объекты, принадлежащие определенному пользователю, теперь обозначается термином "схема", более совместимым со стандартом ANSI. Такое изменение подхода следует учитывать не только потому, что компания Microsoft стала на путь перехода от использования понятия "владельца" к применению понятия "схемы", но и в связи с тем, что работа в этом направлении продолжается, а это означает, что в будущих проектах отказ от использования схем станет уже невозможным. В версии SQL Server 2005 предусмотрены новые функции, обеспечивающие поддержку применения схем в структуре имен, и даже новые образцы баз данных, которые входят в поставку этой версии (в том числе база данных `AdventureWorks`, которая уже встречалась в данной книге время от времени, а в следующих главах будет применяться более часто), характеризуется широким использованием схем. (По мнению автора, иногда при этом количество применяемых схем даже выходит за пределы разумного.) Кроме того, схемы становятся важной, неотъемлемой частью организации работы, когда приходится сталкиваться с некоторыми другими аспектами применения СУБД SQL Server, такими как приложение Notification Services.

Теперь рассмотрим, что представляют собой схемы и как они действуют.

В предыдущих выпусках СУБД SQL Server применялось понятие "владельца" (в той трактовке, которая была принята в то время). По сути толкование этого понятия соответствовало буквальному смыслу слова. Под владельцем подразумевался пользователь, которому "принадлежит" объект, а регистрационное имя этого пользователя включалось в полное уточненное имя объекта. Как правило, владельцем считался либо пользователь, создавший объект, либо владелец базы данных (для обозначения пользователей последней категории чаще используется имя `dbo` – сокращение от database owner; описание категории пользователей `dbo` будет приведено ниже). А начиная с версии SQL Server 2005 распределение объектов по разным частям базы данных осуществляется подобным образом, но сами объекты обозначаются как принадлежащие схеме, а не владельцу. Понятие владельца связано с одной конкретной учетной записью, а с появлением понятия схемы возникла возможность обеспечивать совместный доступ к одной и той же части базы данных для нескольких учетных записей, а также предоставлять одной учетной записи права доступа к нескольким схемам.

По умолчанию создавать объекты в базе данных имеют право только пользователи, которые относятся к системной роли `sysadmin`, а также пользователи, относящиеся к таким ролям базы данных, как `db_owner` или `db_ddladmin`.

В версии SQL Server 2005 предусмотрено гораздо больше системных ролей и ролей базы данных по сравнению с упомянутыми выше. Роли имеют множество прав, предоставленных им с учетом того, для чего может использоваться эта роль. Назначение определенному пользователю конкретной роли равносильно предоставлению этому пользователю возможности пользоваться всеми правами, которые предоставлены данной роли.

Права на создание объектов базы данных и системных объектов некоторых типов могут быть также предоставлены отдельным пользователям. А если такие пользователи действительно создают некоторый объект, то по умолчанию такой объект ста-

новится принадлежащим той схеме, которая рассматривается как принадлежащая по умолчанию учетной записи этого пользователя.

> Не следует без разбора применять какое-то средство лишь потому, что есть такая возможность! Предоставление прав на создание объектов базы данных отдельным пользователям с помощью оператора CREATE нельзя назвать иначе, чем непродуманным решением. Дело в том, что из-за этого становится буквально невозможно проследить за тем, кто, когда и по какой причине создал тот или иной объект. Короче говоря, следует ограничиваться предоставлением доступа к оператору CREATE либо только учетной записи sa, либо представителям привилегированной роли sysadmin или db_owner.

Схема dbo, применяемая по умолчанию

Представителем роли dbo, или "владельцем базы данных", считается любой пользователь, создавший базу данных. Поэтому любые объекты, создаваемые пользователем с такой ролью в базе данных, присваиваются схеме, обозначенной именем dbo, а не именем самого этого пользователя.

Например, предположим, что некоторый пользователь повседневно работает с базой данных, имеет регистрационное имя MySchema и обладает правом на создание таблиц, CREATE TABLE, в какой-то конкретной базе данных. Таблица MyTable, созданная этим пользователем, приобретает имя объекта, уточненное именем владельца, MySchema.MyTable. Обратите внимание на то, что эта таблица имеет конкретного владельца, поэтому любой пользователь, отличный от него (напомним, что владельцем является пользователь MySchema), для доступа к таблице MySchema.MyTable должен указать имя, уточненное именем владельца, для того, чтобы СУБД SQL Server могла определить, к чему относится имя таблицы.

А теперь предположим, что имеется также пользователь с регистрационным именем Fred и что Fred является владельцем базы данных (а не просто одним из представителей роли db_owner). Если пользователь Fred создаст таблицу MyTable с помощью оператора CREATE, идентичного тому, который применялся пользователем MySchema, то данная таблица приобретет имя, уточненное именем владельца, dbo.MyTable. Кроме того, оказывается, что dbo также является владельцем, рассматриваемым как таковой по умолчанию, поэтому любой пользователь может обращаться к этой таблице просто как к MyTable.

Следует отметить, что пользователи с учетной записью sa (или представители роли sysadmin) всегда связаны псевдонимами с ролью dbo. Это означает, что пользователь sa всегда имеет полный доступ к любой базе данных (независимо от того, кто фактически является владельцем базы данных), как если бы он имел роль dbo применительно к этой базе данных, а любые объекты, созданные после регистрации в учетной записи sa, показывают, что по своей принадлежности они относятся к роли dbo. С другой стороны, объекты, созданные представителями роли базы данных db_owner, не попадают по умолчанию во владение роли dbo, рассматриваемой как название схемы, применяемой по умолчанию; такие объекты присваиваются той схеме, которая определена как применяемая по умолчанию для данного, конкретного пользователя (таковым может быть любой пользователь). В это трудно поверить, но дела обстоят именно так!

Автор во время личных встреч со старыми друзьями из компании Microsoft убедился в том, что и на них подействовали призывы перейти к использованию схем, поэтому они приветствуют такие изменения. Я также готов воспользоваться возможностями применения схем, но главным образом потому, что они позволяют в определенной степени упростить работу, а не по тем причинам, что на них основаны такие программные средства, какими нужно было бы немедленно воспользоваться.

В результате введения схем сложность организации базы данных возрастает, независимо от того, какая структура при этом используется. Безусловно, схемы позволяют учесть в проекте возможность распределения объектов с учетом организационной структуры предприятия, но обычно такой же цели можно достичь другими способами, позволяющими создать базу данных, гораздо более удобную в эксплуатации. Кроме того, несмотря на поддержку понятия схем в стандарте ANSI, эта поддержка не обеспечивается полностью одинаково во всех основных программных продуктах реляционных СУБД. Это означает, что использование схем связано с необходимостью учитывать различия между СУБД, если должна осуществляться разработка кода, способного поддерживать несколько платформ.

`DatabaseName` — компонент схемы именования, соответствующий имени базы данных

Следующим компонентом в полностью уточненном имени таблицы, предусмотренным соглашением об именовании, является имя базы данных. Дело в том, что иногда возникает необходимость осуществлять выборку данных из базы данных, отличной от применяемой по умолчанию, или текущей базы данных. Кроме того, может действительно потребоваться использовать операцию соединения к данным, хранящимся в разных базах данных. Такая возможность предоставляется благодаря использованию имен таблиц, уточненных именем базы данных. Например, если пользователь зарегистрировался и указал `AdventureWorks` в качестве текущей базы данных, но ему требуется также обратиться к таблице `Orders` в базе данных `Northwind`, то он имеет возможность указать имя требуемой таблицы в формате `Northwind.dbo.Orders`. К тому же схема `dbo` применяется по умолчанию, поэтому для обозначения этой таблицы можно использовать имя `Northwind..Orders`. Итак, если таблица `MyTable` базы данных `MyDatabase` принадлежит схеме `MySchema`, то имя этой таблицы можно указать как `MyDatabase.MySchema.MyTable`. Напомним, что текущая база данных (определенная с помощью команды `USE` или указанная в раскрывающемся списке, если используется программа SQL Server Management Console) всегда является применяемой по умолчанию, поэтому если требуются только данные из текущей базы данных, то нет необходимости включать имя базы данных в полностью уточненное имя таблицы.

`ServerName` — компонент схемы именования, соответствующий имени сервера

В имени таблицы можно не только указывать другую базу данных, отличную от текущей, на сервере, к которому подключен пользователь, но и устанавливать так называемую "связь" с другим сервером. Установление связи с другими серверами дает возможность применять операцию соединения к данным, хранящимся на разных серверах одного и того же типа и даже на серверах разных типов (SQL Server, Oracle, DB2, Access и т.д.), а при использовании провайдера OLE DB — на серверах практически любых типов. Дополнительная информация о связанных серверах приведена

в одной из следующих глав настоящей книги, но на данный момент достаточно понять, что применение имени сервера позволяет ввести еще один уровень в иерархию именования таблиц, который дает возможность обращаться к различным серверам, и что этот уровень используется во многом аналогично тому, что и уровни, соответствующие обозначениям базы данных и владельца.

А теперь в качестве иллюстрации к использованию этого компонента дополним предыдущий пример. Если требуется осуществить выборку информации с сервера, для которого создана связь, называемая `MyServer`, из таблицы `MyTable` базы данных `MyDatabase`, принадлежащей схеме `MySchema`, то полностью уточненное имя таблицы принимает вид `MyServer.MyDatabase.MySchema.MyTable`.

Значения компонентов полностью уточненного имени таблицы, применяемые по умолчанию

Еще раз рассмотрим и уточним, какие значения применяются по умолчанию на каждом уровне иерархии именования таблицы справа налево, как описано ниже.

- `ObjectName`. Значение по умолчанию отсутствует; имя объекта должно быть задано явно.
- `Ownership`. Этот компонент может оставаться незаданным; в таком случае СУБД вначале предпринимает попытку определить значение компонента с использованием имени текущего пользователя, а затем, если объект с указанным именем не существует в той схеме, владельцем которой является текущий пользователь, то предпринимается попытка найти объект в схеме, владельцем которой служит пользователь `dbo`.
- `DatabaseName`. Этот компонент имени также может быть исключен, при условии, что не задан `ServerName`. Если же компонент `ServerName` применяется, то для СУБД SQL Server должен быть указан и компонент `DatabaseName` (в СУБД других типов могут предъявляться другие требования, в зависимости от конкретного типа СУБД).
- `ServerName`. В качестве этого компонента должно быть указано имя связанного сервера, но чаще всего компонент остается не заданным, в результате чего СУБД SQL Server применяет по умолчанию тот сервер, на котором зарегистрировался пользователь.

Если нет необходимости указывать владельца объекта, но все еще требуется предоставить информацию, касающуюся базы данных или сервера, то все равно необходимо предусмотреть использование лишней точки (`.`) в той позиции, где должен быть указан владелец. Например, если пользователь зарегистрировался на локальном сервере и указал, что будет использовать базу данных `Northwind`, а затем решил обратиться к таблице `Orders` базы данных `Northwind` на связанном сервере `MyOtherServer`, то он может указать имя этой таблицы как `MyOtherServer.Northwind..Orders`. Поскольку в этом имени не задан компонент, определяющий конкретное имя владельца, то предполагается, что для определения владельца требуемого объекта (в данном случае таблицы `Orders`) в качестве компонента должен использоваться либо идентификатор пользователя, с которым он зарегистрировался на связанном сервере, либо имя `dbo` (причем проверка возможных применяемых имен производится именно в указанном порядке).

Оператор CREATE

К сожалению, создание объектов базы данных относится к числу наиболее трудных задач администрирования данных. Прежде всего для успешного создания объектов необходимо тщательно соблюдать установленный синтаксис. Оператором создания объектов является CREATE.

Рассмотрим полную структуру оператора CREATE, начиная с его наиболее обобщенной формы. Эта форма одинаково присуща операторам CREATE, предназначенным для создания всевозможных объектов базы данных, а различия обнаруживаются только в деталях. Первая часть оператора CREATE всегда выглядит таким образом:

```
CREATE <object type> <object name>
```

За этой частью следуют дополнительные уточнения, состав которых зависит от типа создаваемого объекта.

Оператор CREATE DATABASE

Предположим, что необходимо создать базу данных Accounting, которая в дальнейшем будет использоваться для создания таблиц. Наиболее простая синтаксическая форма оператора CREATE DATABASE выглядит, как в приведенном выше примере:

```
CREATE DATABASE <database name>
```

> Следует отметить, что к вновь созданному объекту никто не имеет доступа, кроме создавшего его пользователя, администратора базы данных и владельца базы данных (если созданным объектом была база данных, то таковым становится создавший ее пользователь). Это позволяет создавать объекты, беспрепятственно выполнять всю необходимую корректировку и настройку и только после этого предоставлять доступ к вновь созданным объектам другим пользователям.
>
> Необходимо также указать, что оператор CREATE может использоваться только для создания объектов базы данных на локальном сервере (попытка указать конкретное имя сервера оканчивается неудачей).

Если оператор CREATE используется в приведенной выше форме, это приводит к созданию базы данных, полностью аналогичной базе данных model (напомним, что база данных model рассматривалась в главе 1). Тем не менее в действительности всегда требуется ввести какие-либо дополнительные уточнения, касающиеся структуры базы данных. Для этой цели применяется полный синтаксис оператора CREATE DATABASE:

```
CREATE DATABASE <database name>
[ON [PRIMARY]
   ([NAME = <'logical file name'>,]
    FILENAME = <'file name'>
    [, SIZE = <size in kilobytes, megabytes, gigabytes, or terabytes>]
    [, MAXSIZE = size in kilobytes, megabytes, gigabytes, or terabytes>]
    [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terabytes|percentage>])]
 [LOG ON
   ([NAME = <'logical file name'>,]
```

```
    FILENAME = <'file name'>
    [, SIZE = <size in kilobytes, megabytes, gigabytes, or terabytes>]
    [, MAXSIZE = <size in kilobytes, megabytes, gigabytes, or terabytes>]
    [, FILEGROWTH = <kilobytes, megabytes, gigabytes, or terabytes|percentage>])]
[ COLLATE <collation name> ]
[ FOR ATTACH [WITH <service broker>]|
  FOR ATTACH_REBUILD_LOG| WITH DB_CHAINING ON|OFF | TRUSTWORTHY ON|OFF]
[AS SNAPSHOT OF <source database name>]
[;]
```

Следует учитывать, что некоторые опции в этой синтаксической структуре являются взаимоисключающими (например, если применяется ключевое слово FOR ATTACH, то большинство опций, отличных от тех, которые служат для указания местонахождения файлов, становятся недействительными). Очевидно, что синтаксическая структура оператора создания базы данных является довольно сложной, поэтому рассмотрим ее по частям.

Ключевое слово ON

Ключевое слово ON используется для определения местонахождения файла, в котором хранятся данные, и для определения аналогичной информации, касающейся места хранения журнала. Важно отметить, что в первом случае указано также ключевое слово PRIMARY, которое означает, что за ним следует первичная (или основная) файловая группа, предназначенная для физического хранения данных. Для хранения данных можно также предусмотреть использование так называемых вторичных файловых групп, но описание их применения выходит за рамки настоящей книги. Однако на данный момент достаточно придерживаться такой предусмотренной по умолчанию структуры базы данных, в которой вся необходимая информация хранится в одном файле.

> СУБД SQL Server позволяет использовать для хранения базы данных несколько файлов. Более того, эта СУБД позволяет объединять файлы базы данных в логические группировки, называемые *файловыми группами*. Понятия, связанные с применением файловых групп, являются довольно сложными и не рассматриваются в настоящей книге.

Ключевое слово NAME

Это ключевое слово, означающее "имя", нельзя трактовать буквально, поскольку оно определяет не физическое, а логическое имя файла базы данных, заданное в операторе создания базы данных; иными словами, это ключевое слово задает имя, которое будет использоваться в программном обеспечении СУБД SQL Server для ссылки на файл базы данных. Пользователь применяет указанное имя, если ему требуется изменить размеры (увеличить или уменьшить) базы данных и (или) файла.

Ключевое слово FILENAME

Данное ключевое слово полностью соответствует своему буквальному смыслу — оно обозначает физическое имя действительного файла операционной системы на жестком диске. Именно в этом файле будут храниться данные или журнал (в зависимости от того, в каком разделе оператора CREATE DATABASE задано имя файла). В параметре, обозначенном с помощью этого ключевого слова, также могут исполь-

зоваться значения, предусмотренные по умолчанию (при условии, что в операторе создания базы данных применяется простой синтаксис, который был описан в первую очередь), зависящие от того, относится ли ключевое слово к самой базе данных или к журналу. По умолчанию указанный файл будет находиться в подкаталоге `\Data` главного каталога `Program Files\MSSQL.1\MSSQL` (или того каталога, который был указан в качестве главного каталога SQL Server, если имя главного каталога было изменено во время инсталляции). Если речь идет о физическом файле базы данных, то для него применяется такое же имя, как и для базы данных, с расширением `.mdf`. Если же речь идет о журнале, то рассматриваемый файл принимает такое же имя, как и файл базы данных, но приобретает суффикс `_Log` и расширение `.ldf`. Если решено явно задать имя файла, то допускается указывать другие расширения, но автор настоятельно рекомендует придерживаться заданных по умолчанию значений `mdf` (база данных) и `ldf` (журнал). Дополнительно отметим, что вторичные файлы имеют по умолчанию расширения `.ndf`.

Следует учитывать, что параметр `FILENAME` может рассматриваться как необязательный только при том условии, если применяется чрезвычайно простой синтаксис оператора создания базы данных (с помощью которого новая база данных создается на основе базы данных `model`), описанный в данной главе в первую очередь. А если предоставляется какая-либо дополнительная информация, то необходимо и явно задать имя файла, и обязательно ввести полный путь к файлу.

Ключевое слово SIZE

Ключевое слово `SIZE` не представляет собой что-то особенно сложное. Оно определяет то, что соответствует его смыслу, – размер базы данных. По умолчанию размер задается в мегабайтах, но размеры могут быть заданы в килобайтах с использованием обозначения `KB` вместо `MB` после числового значения размера. Если же база данных должна быть значительно больше обычного, то могут применяться обозначения `GB` (гигабайты) или даже `TB` (терабайты). Следует учитывать, что это значение не должно быть меньше размера базы данных `model`, а также должно быть указано целым числом (без десятичной точки и дробной части), поскольку в противном случае будет получено сообщение об ошибке. Если значение параметра `SIZE` не будет задано, то создаваемая база данных первоначально примет такие же размеры, как и база данных `model`.

Ключевое слово MAXSIZE

Ключевое слово `MAXSIZE` также почти полностью соответствует своему буквальному смыслу (т.е. определяет максимальный размер базы данных), но применяется немного иначе по сравнению с параметром `SIZE`. Дело в том, что в СУБД SQL Server предусмотрен специальный механизм, позволяющий автоматически выделять дополнительное дисковое пространство для базы данных (если потребуется ее увеличение) по мере необходимости. Параметр `MAXSIZE` определяет максимальный размер, до которого может вырасти база данных. И в этом случае по умолчанию заданное число рассматривается как определяемое в мегабайтах, `MB`, но, подобно параметру `SIZE`, могут применяться суффиксы `KB`, `GB` или `TB` для задания других единиц измерения размера. Небольшим отличием этого параметра от параметра `SIZE` является то, что для него не предусмотрено четко заданное значение, применяемое по умолчанию. Таким образом, если значение параметра `MAXSIZE` не задано, такая ситуация рассма-

тривается как отсутствие ограничения по максимуму, поэтому максимум практически достигается только после того, как на жестком диске не остается больше места для расширения базы данных.

После достижения базой данных размеров, заданных в параметре MAXSIZE, пользователи начинают получать сообщения об ошибках с указаниями на то, что применяемые ими операции вставки данных не могут быть выполнены. А если максимального размера достигает журнал, то утрачивается возможность осуществлять в базе данных какие-либо действия, регистрируемые в журнале (а таковым является большинство действий). Сам автор рекомендует задавать так называемые **предупреждения**. Предупреждения можно использовать для получения сообщений о возникновении определенных условий (например, связанных с тем, что база данных или журнал почти полностью заполнились). Способы создания предупреждений будут описаны в главе 19.

Кроме того, автор рекомендует всегда задавать в операторе создания базы данных значение MAXSIZE и выбирать его таким образом, чтобы оно было по меньшей мере на несколько мегабайт меньше по сравнению с тем, которое приводит к переполнению диска. Важность этой рекомендации обусловлена тем, что после полного заполнения диска могут возникнуть ситуации, в которых не удастся больше зафиксировать в базе данных какую-либо информацию, предназначенную для постоянного хранения. А если завершаются неудачей попытки СУБД отвести больше места для хранения журнала, то результаты могут оказаться катастрофическими. Кроме того, даже в операционной системе могут иногда возникать проблемы, если дисковое пространство полностью исчерпывается.

Если же есть необходимость разместить на одном диске несколько баз данных, то нельзя допускать, чтобы в процессе расширения каждая из баз данных могла занять полный объем диска за вычетом нескольких мегабайт, поскольку при этом не исключается возможность полного заполнения диска (если будет происходить расширение всех баз данных).

Ключевое слово FILEGROWTH

Итак, параметр SIZE определяет начальный размер базы данных, параметр MAXSIZE позволяет указать, насколько больших размеров может достичь в конечном итоге файл базы данных, а параметр FILEGROWTH определяет, сколько этапов расширения потребуется, чтобы размеры файла базы данных достигли этого максимума. В качестве значения параметра FILEGROWTH задается число, указывающее величину в байтах, килобайтах (KB), мегабайтах (MB), гигабайтах (GB) или терабайтах (TB), на которую могут каждый раз увеличиваться размеры файла базы данных. Еще один вариант состоит в том, что может быть задано значение в процентах, на которое допускается увеличение размера файла базы данных. При использовании этого варианта размер увеличивается на указанный процент от текущего размера файла базы данных. Таким образом, если указано, что файл базы данных с первоначальными размерами 1GB должен увеличиваться, согласно значению FILEGROWTH, на 20%, то в первый раз в результате увеличения он увеличится до 1.2GB, во второй раз — до 1.44GB и т.д.

Ключевое слово LOG ON

Опция LOG ON позволяет определить характеристики конкретного набора файлов, в которые должна записываться информация журнала, а также указать, где имен-

но должны находиться файлы этого набора. Если данная опция не предусмотрена, то СУБД SQL Server создает журнал в единственном файле и определяет для этого файла по умолчанию размер, равный 25% от размера файла данных. Во всех прочих отношениях файл журнала имеет такие же параметры спецификации файла, как и файл самой базы данных.

Настоятельно рекомендуется хранить файлы журнала на жестком диске, отличном от того, на котором находятся основные файлы данных. Это позволяет избежать конфликтов между операциями ввода-вывода, применяемыми к файлам журнала и файлам базы данных, находящимися на одном и том же диске, а также предусмотреть дополнительную защиту на случай отказа одного из жестких дисков.

Ключевое слово COLLATE

Ключевое слово `COLLATE` имеет отношение к проблемам выбора порядка сортировки, чувствительности к регистру, а также зависимости от выбранной кодовой страницы. Применяемый по умолчанию способ упорядочения (collation) задается при инсталляции СУБД SQL Server, но указанное значение можно перекрывать на уровне базы данных (а также, как будет описано ниже, на уровне столбца).

Ключевое слово FOR ATTACH

Опция `FOR ATTACH` может использоваться для присоединения существующего набора файлов базы данных к текущему серверу. Рассматриваемые файлы должны составлять часть базы данных, которая была в какой-то момент отсоединена от базы данных должным образом с использованием процедуры `sp_detach_db`. При обычных условиях присоединение набора файлов в базе данных осуществляется с помощью процедуры `sp_attach_db`, но команда `CREATE DATABASE` с ключевым словом `FOR ATTACH` имеет свое преимущество в том, что позволяет обеспечить доступ к файлам, количество которых может превышать 32 тысячи, тогда как возможности процедуры `sp_attach_db` ограничиваются всего лишь 16 файлами.

Если применяется ключевое слово `FOR ATTACH`, то должна быть также задана информация о местонахождении файлов в части `ON PRIMARY` оператора создания базы данных. Другие части списка параметров оператора `CREATE DATABASE` могут быть исключены, при условии, что файлы, присоединяемые к базе данных, находятся в том же пути к файлам, в котором они находились при первоначальном отсоединении.

Ключевое слово WITH DB CHAINING ON|OFF

Тема, связанная с использованием ключевого слова `WITH DB CHAINING ON|OFF`, является довольно сложной, особенно для тех, кто лишь недавно приступил к изучению СУБД SQL Server. Поэтому в данном разделе приведены только начальные сведения по этой теме.

Как было указано выше, в действительности понятие "схемы" в предыдущих версиях СУБД SQL Server не использовалось. Вместо этого для определения принадлежности объектов служило такое понятие, как "владелец". Подобная организация работы была несовершенной и, в частности, приводила к таким ситуациям, что образовывались так называемые "цепочки владельцев". Наличие цепочки владельцев характеризуется тем, что, допустим, пользователь `A` является владельцем некоторого объекта, после чего пользователь `B` становится владельцем объекта, который зависит

от другого объекта, принадлежащего пользователю A. Не исключена также возможность, что еще какие-то пользователи будут продолжать создавать объекты, зависящие от объектов других пользователей, что приведет к возникновению сложного переплетения проблем, связанных с предоставлением прав доступа к объектам.

Параметр WITH DB CHAINING ON|OFF определяет, должны ли поддерживаться такие цепочки владельцев, которые пересекают границы между базами данных (когда объект пользователя A находится в базе данных DB1, а зависящий от него объект пользователя B — в базе данных DB2). Если этот параметр имеет значение ON, то цепочки владельцев, охватывающие несколько баз данных, становятся применимыми, а если задано значение OFF, то такие цепочки не поддерживаются. Следует избегать создания цепочек владельцев как настоящего бедствия; для базы данных они и являются настоящим бедствием!

Ключевое слово TRUSTWORTHY

Ключевое слово TRUSTWORTHY представляет собой новое средство, которое позволяет ввести дополнительный уровень защиты доступа к системным ресурсам и файлам вне контекста СУБД SQL Server. Например, предположим, что эксплуатируется какая-либо сборка (assembly) .NET, в которой применяются операции доступа к файлам в сети. В таком случае необходимо обозначить базу данных, к которой относится эта сборка, как принадлежащую к категории Trustworthy (Заслуживающая доверия).

По умолчанию этому параметру по соображениям защиты присвоено значение OFF. Прежде чем задать для него значение ON, необходимо полностью разобраться в том, что вы делаете и в каких целях.

Создание базы данных

После этой предварительной подготовки мы можем приступить к созданию базы данных. Ниже приведен оператор, предназначенный для создания базы данных, но, как уже было сказано, сама база данных — это только один из многих объектов, которые должны быть созданы для того, чтобы база данных стала полностью работоспособной.

```
CREATE DATABASE Accounting
ON
  (NAME = 'Accounting',
   FILENAME = 'c:\Program Files\Microsoft SQL Server\MSSQL.1\mssql\data\
AccountingData.mdf',
   SIZE = 10,
   MAXSIZE = 50,
   FILEGROWTH = 5)
LOG ON
  (NAME = 'AccountingLog',
   FILENAME = 'c:\Program Files\Microsoft SQL Server\MSSQL.1\mssql\data\
AccountingLog.ldf',
   SIZE = 5MB,
   MAXSIZE = 25MB,
   FILEGROWTH = 5MB)

GO
```

Начиная с этого момента мы можем также приступить к изучению некоторых вспомогательных программ, позволяющих получить более подробную информацию о базе данных, которые входят в состав программного обеспечения СУБД SQL Server. В главе 4 показан пример использования утилиты sp_help, но в данном случае попытаемся применить команду sp_helpdb. Утилита sp_helpdb специально предназначена для получения информации о структуре базы данных и часто предоставляет намного больший объем сведений по сравнению с другими утилитами, если требуется скорее узнать о характеристиках самой базы данных, чем о содержащихся в ней объектах. Утилита sp_helpdb принимает один параметр — имя базы данных:

```
EXEC sp_helpdb 'Accounting'
```

В результате выполнения этой команды фактически формируются два результирующих набора. Первый из них, как показано в табл. 5.1, содержит информацию, объединяющую сведения не только о данных, хранящихся в базе данных, но и о ее журналах.

Таблица 5.1. Информация о данных и журналах базы данных

name	db_size	owner	dbid	created	status	compatibility_level
Accounting	15.00 MB	sa	9	May 28 2005	Status=ONLINE, Updateability=READ_WRITE, UserAccess=MULTI_USER, Recovery=FULL, Version=598, Collation=SQL_Latin1_General_CP1_CI_AS, SQLSortOrder=52, IsAutoCreateStatistics, IsAutoUpdateStatistics, IsFullTextEnabled	90

Приведенные в табл. 5.1 данные относятся к определенному компьютеру, а на другом компьютере могут быть получены другие данные. Например, значение DBID зависит от того, сколько баз данных хранится на компьютере и в какой последовательности эти базы данных были созданы. А состав сообщений о статусе (status) зависит от того, какие опции сервера были заданы во время создания баз данных, а также от того, какие опции базы данных были изменены в процессе ее эксплуатации.

Следует отметить, что свойство db_size представляет собой суммарную величину, объединяющую в себе размер базы данных и размер журнала.

Второй результирующий набор, пример которого приведен в табл. 5.2, содержит конкретные сведения о различных файлах, из которых состоит база данных, включая их текущие размеры и параметры, от которых зависит увеличение размеров.

После создания таблиц и вставки данных размеры базы данных начинают увеличиваться автоматически по мере необходимости.

Таблица 5.2. Информация о файлах базы данных

name	fileid	Filename	file-group	size	maxsize	growth	usage
Accounting	1	C:\Program Files\ Microsoft SQL Server\mssql\data\ AccountingData.mdf	PRIMARY	10240 KB	51200 KB	5120 KB	data only
AccountingLog	2	C:\Program Files\ Microsoft SQL Server\mssql\data\ AccountingLog.ldf	NULL	5120 KB	25600 KB	5120 KB	log only

Оператор CREATE TABLE

Первая часть оператора создания таблицы во многом напоминает по своей структуре оператор создания любого объекта, приведенный выше в данной главе. Рассмотрим еще раз общую структуру оператора создания объекта:

```
CREATE <object type> <object name>
```

Поскольку в данном разделе речь идет о создании таблиц, то указанная структура может быть уточнена следующим образом:

```
CREATE TABLE Customers
```

Тем не менее при использовании оператора CREATE DATABASE можно было действительно ограничиться только подобными первыми тремя ключевыми словами, в результате чего была бы создана база данных, соответствующая тем параметрам, применяемым по умолчанию, которые определены в базе данных model. Но применительно к таблицам подобный образец отсутствует, поэтому необходимо задать все подробные сведения, касающиеся состава столбцов, применяемых в таблице типов данных и специальных операций.

В связи с этим рассмотрим более подробную синтаксическую структуру оператора создания таблицы:

```
CREATE TABLE [database_name.[owner].]table_name
(<column name> <data type>
[[DEFAULT <constant expression>]
   |[IDENTITY [(seed, increment) [NOT FOR REPLICATION]]]]
   [ROWGUIDCOL]
   [COLLATE <collation name>]
   [NULL|NOT NULL]
   [<column constraints>]
   |[column_name AS computed_column_expression]
   |[<table_constraint>]
   [,...n]
)
[ON {<filegroup>|DEFAULT}]
[TEXTIMAGE_ON {<filegroup>|DEFAULT}]
```

Вполне очевидно, что синтаксис этого оператора является довольно сложным, несмотря на то, что ради упрощения в этом определении исключены некоторые разделы. Как обычно, рассмотрим составные части синтаксического определения последовательно, начиная со второй строки (описание первой строки уже было приведено).

Имена таблиц и столбцов

Следует отметить, что от выбора имен объектов зависит очень многое. Об этом уже шла речь в главе 2 и было сказано, что мы еще будем возвращаться к этому вопросу в настоящей книге.

Вообще говоря, для именования таблиц и столбцов применяются такие же правила, которые относятся ко всем другим объектам базы данных. В документации СУБД SQL Server они именуются **правилами выбора идентификаторов**, а в данной книге речь о них шла в конце главы 1. Эти правила именования объектов довольно просты, но в данном разделе основное внимание уделено не описанию принципов, используемых в СУБД SQL Server для определения того, какие имена являются допустимыми и недопустимыми, а вопросам такого выбора имен таблиц и столбцов, чтобы они были удобными и имели смысл.

На практике разработано бесчисленное количество различных "стандартов", касающихся того, как следует именовать объекты базы данных, особенно таблицы и столбцы. Ниже приведены довольно простые правила, рекомендуемые автором.

- В каждом слове, которое входит в состав имени, перевести в верхний регистр первую букву и использовать нижний регистр для написания остальных букв.
- Выбирать короткие имена, которые вместе с тем имеют достаточную длину для того, чтобы можно легко понять их назначение.
- Ограничить использование аббревиатур. Единственный приемлемый способ использования аббревиатур состоит в том, чтобы выбранные аббревиатуры мог распознать любой пользователь. В качестве примеров аббревиатур, используемых автором, достаточно привести "ID" (сокращение от identification – идентификация или идентификатор), "No" (сокращение от number – количество) и "Org" (сокращение от organization – организация). Иногда в связи с необходимостью лимитирования длины имен приходится широко пользоваться аббревиатурами, но следует всегда учитывать, что первым и наиболее важным требованием к именам всегда остается то, что они должны быть понятными и запоминающимися.
- При создании таблиц, предназначенных для обеспечения связи между другими таблицами (и обычно называемых соединительными, или ассоциативными, таблицами), необходимо включить в имя новой таблицы имена всех родительских таблиц. Например, предположим, что имеется база кинематографических данных, в которой хранятся данные о звездах кинематографа и фильмах, в которых они снимались. Между фамилиями звезд и названиями фильмов имеется связь "многие ко многим". Поэтому для оформления связи между таблицами `Movies` и `Stars` применяется отдельная таблица, которая должна иметь имя `MovieStars`.
- Если имя объекта базы данных должно состоять из двух и более слов, не следует вводить между словами какие-либо разделители (все слова должны записываться слитно). Для распознавания границ между словами достаточно воспользоваться тем фактом, что первая буква каждого слова является прописной, а остальные буквы – строчными.

Если бы автор глубже коснулся этой темы, то ему пришлось бы спорить с другими специалистами в области баз данных по поводу проблем именования и он не смог бы вообще закончить эту книгу. Например, очень многие специалисты считают, что следует разделять слова, из которых состоит имя объекта, символами подчеркивания (_).

Но автор отказывается от такого способа именования по некоторым простым причинам, которые связаны с тем, что из-за применения символа подчеркивания возникает целый ряд проблем, которые описаны ниже.

- Прежде всего, чтобы ввести символ подчеркивания, необходимо снять пальцы с правильной позиции на клавиатуре. Многие люди испытывают при этом затруднения, в связи с чем возникают многочисленные опечатки.
- Далее, в документации нередко приходится проводить линию под именем таблицы или столбца (применять стиль подчеркивания), а некоторые шрифты не позволяют обнаружить символы подчеркивания на фоне линии, которой подчеркнут текст, поскольку эти символы сливаются с линией подчеркивания; это приводит к путанице и дополнительным ошибкам.
- Наконец, из-за применения символов подчеркивания объем вводимого кода увеличивается (автор согласен с тем, что это замечание — удар ниже пояса).

В версии SQL Server 7.0 появилась также возможность разделять слова в имени с использованием обычного пробела. Но автор уже отметил в первом своем критическом замечании (см. главу 1), что такая возможность не приносит никакой пользы, поскольку применение пробелов в именах представляет собой чрезвычайно плохую практику программирования и приводит к появлению невероятного количества ошибок. В действительности использование пробелов в именах было разрешено в целях стимулирования дальнейшего распространения СУБД Access, но автор еще раз предостерегает тех пользователей, которые решили поддаться на эту приманку, — можно не сомневаться, что возможность применения пробелов была предоставлена с самыми лучшими намерениями, но практика показывает, что появление подобных имен стало причиной многочисленных нарушений в работе баз данных.

Безусловно, приведенный выше перечень рекомендаций не является исчерпывающим; его можно, скорее, назвать популярной версией правил, фактически используемых автором при именовании таблиц. Просто я обнаружил, что применение указанных правил избавляет меня от многих проблем, и надеюсь, что такую же пользу из них извлечет и читатель.

Наиболее важным требованием к выбору имен является соблюдение единообразия. Занимаясь со своими студентами, автор каждый раз предупреждает их заранее, что слово "единообразие" будет повторяться на лекциях снова и снова, но когда речь идет об именовании объектов, то буквально нельзя найти более важного понятия. Приняв на вооружение какое-то правило, которому вы собираетесь следовать, усвойте еще одно правило, которое гласит: каковы бы ни были выбранные вами стандарты, относитесь к ним действительно как к стандартам. Например, если вы решили по какой-то причине применять определенную аббревиатуру, то неизменно используйте ее вместо развернутого термина (причем применяйте ее в одном и том же виде). Независимо от того, какие правила именования вы выбрали, соблюдайте единообразие, применяя эти правила ко всем объектам базы данных. При этом лучше всего подготовить документ с описанием стандартов именования или руководство по стилю, чтобы другие разработчики могли пользоваться такими же правилами, какие применяете вы. Это позволит исключить невероятное количество ошибок, а также ускорить изучение пользователями предоставленной им для работы базы данных.

Типы данных

В настоящей главе этой теме не будет уделено много внимания, поскольку типы данных были подробно описаны в главе 2. Следует лишь отметить, что за именем каждого столбца должно быть сразу же приведено обозначение типа данных, поскольку не предусмотрены типы данных, применяемые по умолчанию.

Ключевое слово DEFAULT

Применение ключевого слова DEFAULT будет рассматриваться более подробно в главе, посвященной ограничениям, а на данный момент достаточно отметить, что это ключевое слово определяет значение, которое должно использоваться при вставке любой строки в качестве значения определенного поля, если для этого поля не предусмотрено значение, заданное пользователем. Если в определении таблицы предусмотрено применение значений, заданных по умолчанию, то оно должно следовать сразу же за обозначением типа данных.

Ключевое слово IDENTITY

При проектировании базы данных понятие **идентификации** является очень важным. В главах, посвященных проектированию, способы использования столбцов идентификации (identity) будут рассматриваться более подробно. А в данном разделе рассмотрим определение понятия столбца идентификации. Если в объявлении столбца указано, что этот столбец является столбцом идентификации, то СУБД SQL Server автоматически присваивает последовательное числовое значение полю, относящемуся к этому столбцу, при вставке каждой строки. Число, с которого в СУБД SQL Server начинается отсчет значений в столбце идентификации, называется **начальным значением**, а величина, на которую это значение увеличивается или уменьшается после вставки каждой строки, называется **шагом**. По умолчанию и начальное значение, и шаг равны 1; практика показывает, что в большинстве проектов эти значения остаются неизменными. Тем не менее, например, можно установить начальное значение, равное 3, и применить шаг, равный 5. В таком случае отсчет начинается с 3, а после ввода каждой строки с этой величиной складывается значение 5, что приводит к получению идентификационных значений 8, 13, 18, 23 и т.д.

Столбец идентификации должен быть числовым, причем на практике такой столбец почти всегда объявляется с типом данных integer или bigint.

Применение столбцов идентификации является довольно несложным — достаточно задать ключевое слово IDENTITY сразу после указания типа данных для столбца. Но возможность использования автоматически наращиваемого идентификационного значения не совместима с применением значения, предусмотренного по умолчанию. Если немного подумать, то такое условие действительно имеет смысл, ведь заданное по умолчанию значение является постоянным, а идентификационное значение после вставки каждой строки увеличивается или уменьшается.

Следует учитывать, что столбец идентификации функционирует последовательно. Это означает, что сразу после того, как будут определены параметры seed (начальное значение) и increment (шаг), неизменно происходит увеличение очередных формируемых значений (или их уменьшение, если в качестве шага задано отрицательное число). При этом не предусмотрено какого-либо автоматизированного механизма, позволяющего возвращаться и заполнять идентификационные значения вместо значений тех строк, которые, возможно, были удалены. Если же в подобных случаях требуется заполнить пустующие поля, то необходимо использовать опцию SET IDENTITY_INSERT ON, которая позволяет остановить

процесс выработки идентификационных значений для операторов вставки, выполняемых в текущем соединении (да, здесь нет ошибки; после задания значения параметра ON выработка и вставка идентификационных значений прекращается; иными словами, параметр ON указывает, что пользователь желает получить возможность вставлять самостоятельно вырабатываемые значения, для чего необходимо отменить использование средств автоматической вставки значений). Тем не менее применение опции SET IDENTITY_INSERT ON может привести к серьезному нарушению в работе, если не соблюдается предельная осторожность или если не контролируются операции вставки в ту же таблицу, выполняемые другими пользователями, поэтому необходимо действовать осмотрительно.

Столбцы идентификации чаще всего используются для выработки все новых и новых значений, которые служат в качестве идентификаторов для каждой строки; иными словами, столбцы идентификации обычно используются для создания первичного ключа таблицы. Но следует учитывать, что столбцы идентификации IDENTITY и столбцы первичного ключа PRIMARY KEY нельзя рассматривать как равнозначные. Дело в том, что столбцы первичного ключа должны содержать уникальные значения, а значения в столбцах идентификации не всегда являются уникальными (например, при использовании столбца IDENTITY допускается и переустанавливать начальное значение, и переходить к использованию обратного отсчета, в результате чего будут повторяться ранее заданные значения). Действительно, значения столбцов IDENTITY часто используются в качестве значений столбцов PRIMARY KEY, но они не обязательно должны служить только для этой цели.

Разработчики, знакомые с СУБД Access, могут обнаружить, что столбцы IDENTITY во многом аналогичны столбцам AutoNumber. Основное различие между ними состоит в том, что СУБД SQL Server обеспечивает немного больший контроль над значениями этих столбцов, чем СУБД Access.

Ключевое слово NOT FOR REPLICATION

На данный момент задача достаточно подробного описания опции NOT FOR REPLICATION является довольно сложной, поэтому рассмотрим лишь ее основные возможности, а остальное описание отложим до той главы, где речь идет о репликации.

Кратко процесс репликации можно определить очень неформально как автоматическое копирование некоторой или всей информации из одной базы данных в другую. Такая другая база данных может находиться на том же физическом компьютере, что и база данных, используемая в качестве оригинала, либо может находиться на удаленном компьютере.

Параметр NOT FOR REPLICATION определяет, следует ли присваивать столбцу, публикуемому (с помощью репликации) в другой базе данных, новое идентификационное значение (вырабатываемое в этой новой базе данных) или сохранять существующее значение. Дополнительные сведения по этой теме будут приведены ниже.

Ключевое слово ROWGUIDCOL

Опция ROWGUIDCOL также связана с репликацией и применительно к столбцу идентификации имеет в основном такое же назначение. Как уже было отмечено, применение

столбца идентификации позволяет предусмотреть удобный способ выработки значений, уникальных для каждой строки, которые поэтому могут использоваться для однозначного обозначения этой строки. Но решение, основанное на применении столбца идентификации, способствует возникновению ошибок, если оно реализовано в такой среде, в которой применяется репликация, или в другой распределенной среде.

В этом можно легко убедиться, представив себе такую ситуацию, что в двух взаимосвязанных объектах из разных баз данных применяются столбцы идентификации, значения в которых изменяются в бо́льшую сторону, начиная от некоторой установленной величины. В таком случае трудно представить себе, что могло бы предотвратить появление одинаковых идентификационных значений в различных базах данных. А если предположить, что в дальнейшем будет предпринята попытка применения такого сценария репликации, в котором все строки, ранее находившиеся в отдельных базах данных, теперь должны перейти в одну базу данных, то из-за наличия повторяющихся значений идентификаторов возникнут весьма значительные сложности. В столбце, предназначенном для однозначной идентификации каждой строки, теперь должны появиться дублирующиеся значения!

С годами сложилось такое общепринятое решение этой задачи, которое предусматривает использование отдельных начальных значений для каждой базы данных, применяемой в качестве источника или получателя реплицированных данных. Например, может быть предусмотрено, чтобы в базе данных `A` отсчет идентификационных значений начинался с `1`, в базе данных `B` — с `10001`, а в базе данных `C` — с `20001`. Таким образом, появляется возможность без опасений публиковать данные всех этих трех баз данных в какой-то четвертой базе данных. Но это не может продолжаться до бесконечности. Как только, допустим, база данных `A` вырастет в объеме и станет включать больше 9999 строк, начнутся большие неприятности.

На это можно было бы возразить, указав, что решением может служить разделение начальных значений не на 10000, а на 100000 или даже на 500000. Но если обмен данными с таблицей происходит достаточно активно, то указанное решение лишь отдаляет неизбежный конец. Именно в связи с этим применяется решение, основанное на ключевом слове `ROWGUIDCOL`.

Столбцы с атрибутом `ROWGUIDCOL` во многом напоминают столбцы с атрибутом `IDENTITY` в том, что обычно используются для однозначной идентификации каждой строки в таблице. А их отличительной особенностью является гораздо бо́льшая длина последовательности значений в столбце `ROWGUIDCOL`, в пределах которой система гарантирует полную уникальность вырабатываемых значений. В СУБД SQL Server при формировании значений `ROWGUIDCOL` применяется не наращивание числовых значений, а выработка так называемых глобально уникальных идентификаторов, или **GUID** (Globally Unique Identifier). Можно условно считать, что значения столбцов идентификации обычно являются уникальными во времени (если параметры, применяемые при выработке этих значений, остаются неизменными), но они не уникальны в пространстве. Поэтому, если эксплуатируются две копии таблиц в разных базах данных, строкам обеих этих таблиц присваиваются повторяющиеся значения в столбцах идентификации. На первых порах в процессе эксплуатации таких баз данных связанные с этим потенциальные проблемы часто не обнаруживаются, а когда настает время объединить строки обеих таблиц в виде одной реплицируемой таблицы, возникают значительные сложности. С другой стороны, значения `GUID` являются уникальными и в пространстве, и во времени.

Значения GUID в наши дни находят в вычислительной технике все более широкое распространение. Например, огромное количество таких значений обнаруживается при проверке реестра операционной системы Windows. Любое значение GUID является 128-битовым, поэтому если оно рассматривается как двоичное значение, выраженное с помощью одного из числовых типов, то требует для своего представления в десятичной форме не меньше 38 знаков. Согласно теоретическим расчетам, если бы новое значение GUID вырабатывалось через каждую секунду, то до появления дубликата числа с такой значностью прошли бы миллионы лет.

Значения GUID вырабатываются на основе использования некоторого сочетания информационных компонентов, причем эти компоненты выбраны таким образом, чтобы каждый из них был уникальным либо в пространстве, либо во времени. А после использования всех этих компонентов в их сочетании формируется значение, для которого с точки зрения статистики гарантируется уникальность в пространстве и времени.

Обычно в программировании для выработки значений GUID используется один из вызовов API-интерфейса Windows, а в языке SQL предусмотрена возможность не только задать для столбца опцию ROWGUIDCOL, но и применить специальную функцию, возвращающую значение GUID; эта функция называется NEWID() и может быть вызвана в любое время.

Ключевое слово COLLATE

Опция COLLATE действует главным образом по такому же принципу, как и в команде CREATE DATABASE; основное различие состоит в том, что в данном случае область действия становится более узкой (поскольку она определяется на уровне столбца, а не на уровне базы данных).

Ключевое слово NULL и (или) NOT NULL

Опция NULL и (или) NOT NULL имеет довольно простое назначение — с ее помощью можно указать, допускает ли столбец, к которому она относится, ввод NULL-значений, или нет. По умолчанию сразу после инсталляции СУБД SQL Server для любого столбца задается значение опции NOT NULL, если только во время инсталляции не будет предусмотрена возможность применения неопределенных значений, или NULL-значений (nullability). Тем не менее имеется очень большое количество параметров настройки конфигурации сервера, которые могут повлиять на эту опцию и изменить способ ее применения. Например, значение этой опции может измениться после задания любого значения с помощью хранимой процедуры sp_dbcmptlevel или установки опций совместимости со стандартом ANSI.

> Автор настоятельно рекомендует явно задавать опцию NULL и (или) NOT NULL для каждого столбца при создании любой таблицы. Эта рекомендация обусловлена тем, что, как уже было сказано, в СУБД SQL Server предусмотрено большое количество различных параметров настройки конфигурации, которые могут повлиять на то, какое значение будет использоваться в системе по умолчанию для определения возможности ввода в столбец неопределенного значения (NULL-значения). Если при формировании оператора создания таблицы разработчик полагается только на такие значения, заданные по умолчанию, то в дальнейшем может обнаружить, что не все его сценарии работают правильно (из-за того что он или кто-то другой изменил один из относящихся к этому параметров настройки, не учитывая всех возможных последствий).

Ограничения столбца

В настоящей книге ограничениям будет посвящена целая глава, поэтому в этом разделе не будем терять времени на их подробное описание. Тем не менее именно сейчас целесообразно рассмотреть, что представляют собой ограничения столбца. Кратко можно отметить, что ограничения столбца представляют собой ограничения и правила, касающиеся данных, которые могут быть вставлены в тот или иной столбец, определяемые на уровне отдельных столбцов.

Например, предположим, что один из столбцов предназначен для хранения числовых обозначений месяцев года. Безусловно, можно определить такой столбец, как имеющий тип `tinyint`, но подобное определение не исключает возможности вставки в него числа 54. Однако в ходе дальнейшей обработки число 54 будет распознано как неправильный компонент даты (ему не соответствует ни один календарный месяц), поэтому необходимо предусмотреть ограничение, которое указывает, что данные в столбце месяцев должны находиться в пределах от `1` до `12`. В следующей главе будет показано, как это можно сделать.

Вычисленные столбцы

В базе данных может быть также предусмотрено использование столбцов, не имеющих собственных данных, но обеспечивающих динамическое вычисление требуемых значений по данным из других столбцов таблицы. На первый взгляд может показаться, что применение таких столбцов является излишеством, поскольку требуемое значение и так можно вычислить в любой момент при выполнении запроса, но в действительности вычисленные столбцы позволяют упростить создание многих приложений.

Например, предположим, что разработчик участвует в создании системы выставления счетов. В этой системе необходимо хранить информацию о количестве проданных единиц товара и ценах, по которым они были проданы. Довольно широко применяется такой подход, при котором предусматриваются столбцы для хранения исходной информации, наряду с еще одним столбцом, в котором хранятся производные значения (произведение цены на количество). Но такой подход приводит к бесполезному расходованию дискового пространства и усложнению задачи сопровождения, связанному с тем, что приходится постоянно следить, чтобы исходные значения и их произведения оставались согласованными друг с другом. А с помощью вычисленного столбца, который определен как произведение цены на количество, можно избавиться от этих проблем, поскольку необходимое значение общей стоимости всегда будет правильно вычисляться автоматически по мере необходимости.

Рассмотрим синтаксис вычисленного столбца:

```
<column name> AS <computed column expression>
```

Первый элемент этого определения, `<column name>`, практически остается таким же, как и в других определениях столбцов, — он задает имя столбца, в котором будут представлены вычисленные значения. Но это — просто псевдоним, который должен использоваться для ссылки на значение, вычисляемое в соответствии с выражением `<computed column expression>`, которое следует за ключевым словом `AS`.

Еще одним элементом этого определения является выражение вычисленного столбца. Это выражение может представлять собой любое обычное выражение, в котором используются литералы и (или) значения столбцов из той же таблицы.

Поэтому применительно к рассматриваемому примеру с ценой и количеством можно определить столбец итоговой цены следующим образом:

```
ExtendedPrice AS Price * Quantity
```

Чтобы показать на примере использование литерала, предположим, что в магазине установлена фиксированная наценка на все товары, которая составляет 20% от стоимости. Применение вычисленного столбца позволяет отслеживать стоимость в одном столбце, а затем вычислять значение столбца продажной цены `ListPrice` с помощью вычисленного столбца:

```
ListPrice AS Cost * 1.2
```

Безусловно, на первый взгляд задача определения вычисленного столбца кажется довольно простой, но необходимо учитывать несколько описанных ниже предостережений и условий их применения.

- В выражении вычисленного столбца нельзя использовать подзапрос, а значения для этого столбца не могут быть взяты из другой таблицы.
- В версиях, предшествующих SQL Server 2000, вычисленные столбцы нельзя использовать в составе любого ключа (первичного, внешнего или уникального); не допускается также применять вычисленные столбцы в сочетании с ограничением, предусмотренным по умолчанию. А что касается версии SQL Server 2005, то вычисленные столбцы допускается использовать в ограничениях (тем не менее в таком случае вычисленный столбец должен быть обозначен как неизменяемый, persisted).
- В версиях, предшествующих SQL Server 2000, обнаруживалась еще одна проблема (но в указанной версии была устранена), которая касалась возможности создания индексов на вычисленных столбцах. В этих версиях можно было создавать индексы на вычисленных столбцах, но для этого приходилось предпринимать специальные действия. Каждое из изменений, касающихся применения вычисленных столбцов, будет рассматриваться более подробно при изучении ограничений в главе 6 и индексации в главе 9.

Более подробные примеры использования вычисленных столбцов приведены ниже в данной главе.

Автора очень удивляет то, что в сообществе пользователей баз данных не ведутся горячие споры в отношении применимости вычисленных столбцов. Правила нормализации данных указывают, что в таблицах не должно быть столбцов с информацией, которая может быть получена из других столбцов, а вычисленные столбцы используются именно для этой цели!

Безусловно, меня вполне устраивает то, что специалисты, придающие слишком большое значение вопросам нормализации, не заостряют особого внимания на проблеме применения вычисленных столбцов, поскольку я положительно оцениваю возможность использования вычисленных столбцов и рассматриваю их как своего рода компромисс. Дело в том, что такие столбцы не требуют дублирования хранимых значений, а их применение не связано с возникновением таких нарушений в работе, что производные значения не согласуются с исходными значениями, ведь при использовании вычисленных столбцов производные значения вычисляются динамически и для этого непосредственно берутся исходные значения. Тем не менее при этом успешно достигается требуемый конечный результат. Но следует отметить, что индексация вычисленного столбца действительно приводит к увеличению затрат памяти на хранение данных (поскольку для индекса выделяется память). Но в этом есть свои преимущества, поскольку увеличивается производительность операций чтения.

Безусловно, вычисленные столбцы не позволяют устранить все проблемы, связанные с получением производных данных, но нет сомнения в том, что они могут оказать неоценимую помощь во многих ситуациях.

Ограничения таблицы

Ограничения таблицы во многом аналогичны ограничениям столбца, поскольку также позволяют определять условия, которым должны соответствовать данные, вставляемые в таблицу. Небольшое различие между ограничениями таблицы и ограничениями столбца обусловлены тем, что ограничения таблицы могут распространяться на несколько столбцов.

Еще раз отметим, что более подробное описание ограничений таблицы будет приведено в главе, посвященной ограничениям. На данный момент достаточно лишь указать, что к примерам ограничений уровня таблицы относятся ограничения PRIMARY и FOREIGN KEY, а также ограничение CHECK.

> На первый взгляд может показаться, что ограничение CHECK нельзя рассматривать как ограничение таблицы, ведь оно применяется на уровне столбца, поскольку влияет на то, будет ли разрешено поместить определенное значение в какой-то конкретный столбец. Тем не менее ограничение CHECK может применяться и на уровне таблицы, и на уровне столбца. Если ограничение CHECK основано исключительно на данных, которые относятся только к одному столбцу, то соответствует определению ограничения столбца. Если же возможность применения ограничения CHECK зависит от нескольких столбцов (и такая форма ограничения CHECK действительно предусмотрена), то его нельзя назвать иначе, чем ограничением таблицы.

Ключевое слово ON

Напомним, что в тех разделах настоящей главы, где речь шла о создании базы данных, было указано, что для хранения базы данных могут быть созданы различные файловые группы. Итак, конструкция ON в определении таблицы и представляет собой способ конкретного указания того, в какой файловой группе (и, соответственно, на каком физическом устройстве) должна находиться таблица. Таким образом, разработчик может поместить определенную таблицу на указанное физическое устройство, но в большинстве случаев конструкция ON остается незаданной, поэтому система помещает создаваемую таблицу в файловую группу, предусмотренную по умолчанию (таковой является группа PRIMARY, если вместо нее не задано какое-то другое значение). Возможность явного распределения таблиц по файловым группам будет рассматриваться более подробно в главе, посвященной настройке производительности.

Ключевое слово TEXTIMAGE_ON

По существу, конструкция TEXTIMAGE_ON аналогична по своему назначению описанной выше конструкции ON, за исключением того, что первая позволяет определить еще одну файловую группу, в которую должна быть перемещена вполне определенная часть таблицы. Конструкция TEXTIMAGE_ON является действительной, только если в определении рассматриваемой таблицы имеется столбец (столбцы) типа text, ntext

или image. Если для таблицы применяется конструкция TEXTIMAGE_ON, то в отдельную файловую группу перемещаются только данные в формате BLOB из столбцов указанного типа, а остальная часть таблицы располагается либо в файловой группе, применяемой по умолчанию, либо в файловой группе, выбранной в конструкции ON.

Распределение базы данных по нескольким файлам и последующее сохранение этих файлов на отдельных физических дисках приводит к существенному повышению производительности. Дело в том, что при такой организации хранения данных в выполнении операций ввода-вывода участвует несколько разных дисков. Более подробное описание данной темы выходит за рамки настоящей книги, но следует иметь в виду, что данный подход заслуживает изучения в том случае, если в процессе эксплуатации базы данных возникают проблемы низкой производительности ввода-вывода.

Создание таблицы

Итак, выше в настоящей главе приведен значительный объем информации, а также даны практические примеры, позволяющие теперь приступить к созданию полноценных таблиц.

В частности, в начале данного раздела был приведен такой синтаксис стандартного оператора CREATE:

```
CREATE <object type> <object name>
```

После этого речь шла о более конкретной форме оператора CREATE, применимого для создания таблицы Customers (это был самый первый пример оператора создания таблицы в настоящей главе):

```
CREATE TABLE Customers
```

Таблица Customers должна стать первой таблицей в базе данных, создаваемой для организации бухгалтерского учета в гипотетической компании. Вопросы проектирования базы данных будут рассматриваться во многих других главах настоящей книги, но вначале будет показано, как приступить к заполнению базы данных на примере использования ряда операторов CREATE TABLE для создания таблиц. В этом разделе должны быть представлены почти все основные понятия, касающиеся формирования таблиц, но некоторые вопросы будут рассмотрены дополнительно в следующих главах. Итак, приступим к созданию первой из нескольких таблиц.

Автор обычно вводит строку USE <database name> перед любым оператором CREATE, в том числе предназначенным для создания таблиц, чтобы гарантировать в результате выполнения данного конкретного сценария создание таблицы в той базе данных, где она действительно должна находиться. А за первой строкой USE следует оператор с уже известным нам форматом, но с тем количеством определений столбцов, которое больше напоминает используемое на практике.

Любой сценарий, предназначенный для регулярного применения в какой-то конкретной базе данных, должен включать команду USE с указанием имени этой базы данных. Соблюдение указанного условия позволяет гарантировать, что операции создания, модификации и удаления объектов действительно будут осуществляться в той базе данных, для которой они предназначены.

Автору неоднократно приходилось становиться жертвой собственной забывчивости, когда он вслепую открывал сценарий и вызывал его на выполнение лишь для того, чтобы затем обнаружить, что текущей оказалась не та база данных, которая требовалась, уничтожены таблицы с именами, совпадающими с указанными в операторе удаления (и при этом потеряны все данные), а вместо существующих таблиц появились таблицы с новой структурой. Кроме того, нередко приходится слышать о том, как разработчики, просматривая базу данных `master`*, обнаруживают в ней несколько посторонних таблиц, которые появились из-за того, что какие-то разработчики выполнили применительно к базе данных* `master` *сценарии CREATE, предназначенные для другой базы данных.*

```
USE Accounting
CREATE TABLE Customers
(
  CustomerNo     int            IDENTITY    NOT NULL,
  CustomerName   varchar(30)                NOT NULL,
  Address1       varchar(30)                NOT NULL,
  Address2       varchar(30)                NOT NULL,
  City           varchar(20)                NOT NULL,
  State          char(2)                    NOT NULL,
  Zip            varchar(10)                NOT NULL,
  Contact        varchar(25)                NOT NULL,
  Phone          char(15)                   NOT NULL,
  FedIDNo        varchar(9)                 NOT NULL,
  DateInSystem   smalldatetime              NOT NULL
)
```

Это — довольно упрощенная таблица по сравнению с теми, которые обычно используются на практике, но мы имеем широкие возможности внести дополнения в ее определение в ходе дальнейшего изложения (и, безусловно, воспользуемся этими возможностями).

После создания таблицы желательно проверить, что она действительно создана и что в ней имеются все предусмотренные столбцы, типы которых соответствуют ожидаемым. Для этой цели можно использовать несколько разных команд, но, по-видимому, наилучшей из них является та, которая станет для вас старой знакомой к концу изучения настоящей книги — `sp_help`. Эта команда имеет простой синтаксис:

```
EXEC sp_help <object name>
```

Чтобы воспользоваться данной командой, указав имя только что созданного объекта таблицы, вызовите на выполнение следующий код:

```
EXEC sp_help Customers
```

Команда `EXEC` имеет два разных назначения. Приведенный здесь вариант ее применения предназначен для выполнения хранимых процедур; в данном случае системной хранимой процедуры. Второй вариант использования этой команды будет описан позже, когда речь пойдет о более сложной тематике выполнения запросов и хранимых процедурах.

С формальной точки зрения допускается возможность выполнения хранимой процедуры просто путем ее вызова (без использования ключевого слова EXEC). Но проблема заключается в том, что такой способ вызова на выполнение приводит к получению единообразных результатов, только если вызов хранимой процедуры задан как первый оператор в пакетном файле любого типа. Безусловно, вместо приведенной выше команды можно было бы просто задать sp_help Customers, но если бы перед этим была предпринята попытка выполнить оператор SELECT, то последующий вызов хранимой процедуры завершился бы аварийно. Отказ от применения ключевого слова EXEC приводит к тому, что хранимые процедуры, вызываемые на выполнение, ведут себя очень непредсказуемо, поэтому необходимо избегать таких вариантов организации работы.

После вызова указанной команды на выполнение обнаруживается, что один за другим формируется несколько результирующих наборов. Поэтому полученная информация включает целый ряд отдельных результирующих наборов, содержащих описанные ниже данные.

- Имя таблицы, схема, тип таблицы (системная или пользовательская) и дата создания.
- Имена столбцов, типы данных, возможность применения неопределенных значений, размер и способ упорядочения.
- Столбец идентификации (если он существует), в том числе информация о начальном значении и шаге.
- Столбец RowGUIDCol (если он существует).
- Информация о файловой группе.
- Имена индексов (если они имеются), типы и включенные в индексы столбцы.
- Имена ограничений (если они имеются), типы и включенные в ограничения столбцы.
- Имена внешних ключей (если они имеются) и относящиеся к ним столбцы.
- Имена всех представлений, относящихся к схеме (дополнительная информация по этой теме приведена в главе 10), которые зависят от таблицы.

Убедившись в том, что таблица создана успешно, перейдем к созданию еще одной таблицы — Employees. Но на этот раз примем другой подход — вначале рассмотрим, для чего должна быть предназначена эта таблица, а затем определим, что можно сделать, чтобы составить код сценария CREATE самостоятельно.

Таблица Employees также должна быть достаточно простой. В частности, в этой таблице должна находиться перечисленная ниже информация.

- Идентификатор служащего. Это значение должно вырабатываться системой автоматически.
- Имя.
- При желании, средний инициал (сокращенное обозначение отчества).
- Фамилия.
- Должность.
- Номер карточки социального обеспечения.

- Зарплата.
- Предыдущая зарплата.
- Величина последнего повышения зарплаты.
- Дата найма на работу.
- Дата окончания договора найма (если таковая известна).
- Руководитель служащего.
- Отдел.

Вначале попытайтесь сформировать компоновку этой таблицы самостоятельно.

Но прежде чем перейти к изучению варианта, приведенного в книге, примите заверения автора в том, что не следует слишком расстраиваться, если предложенная вами компоновка не полностью совпадает с описанной. Дело в том, что и в данном случае действует принцип — сколько голов, столько и мнений, т.е. сколько проектировщиков баз данных, столько и разных проектов баз данных, и все эти проекты начинаются с проектов таблиц. К тому же обычно для любой задачи можно найти несколько разных путей решения. Поэтому важнее всего убедиться в том, что вы действительно смогли учесть все требования, исходящие из поставленной задачи. На этом лирическое отступление заканчивается, поэтому перейдем к рассмотрению одного из способов создания указанной таблицы.

В определении таблицы автор предусмотрел специальный столбец. Значения полей столбца `EmployeeID` должны вырабатываться системой, и поэтому данный столбец превосходно подходит для использования в качестве либо столбца идентификации, либо столбца `RowGUIDCol`. В пользу того или иного варианта может выступать несколько важных факторов, но в данном случае предусмотрено применение столбца `EmployeeID` в качестве столбца идентификации по нескольким описанным ниже причинам.

- Данные, хранящиеся в этом столбце, предназначены для использования обычными людьми (поэтому вряд ли кому-то захочется запоминать значения `GUID`).
- Выбранный вариант приводит к снижению издержек.

Теперь мы можем приступить к формированию оператора создания таблицы:

```
CREATE TABLE Employees
(
  EmployeeID      int         IDENTITY  NOT NULL,
```

Для этого столбца опция `NOT NULL` должна быть выбрана автоматически в связи с использованием опции `IDENTITY`, поскольку в столбце типа `IDENTITY` не допускается наличие NULL-значений. Но следует отметить, что все равно необходимо будет, по всей вероятности, указать опцию `NOT NULL` явным образом, чтобы не зависеть от параметров настройки сервера (если она не задана, то может возникнуть ошибка в том случае, когда по умолчанию допускается использование NULL-значений).

Затем необходимо ввести в определение таблицы столбцы для хранения компонентов имен. Автор обычно допускает применение приблизительно 25 символов для представления имен. Безусловно, большинство компонентов имен имеет гораздо меньшую длину, но на практике слишком часто приходится сталкиваться с тем, что некоторые компоненты имен имеют довольно большую длину (особенно если это имена, соединенные дефисами, которые становятся все более популярными), поэто-

му приходится предусматривать для них лишнее место. Кроме того, для таких столбцов целесообразно использовать строковый тип данных переменной длины по двум описанным ниже причинам.

- Это позволяет не занимать полностью место на внешнем устройстве, отведенное для столбца, если фактически хранимые данные имеют длину меньше объявленной (поскольку система использует незаполненное пространство для других целей).
- Поиск с помощью конструкции `WHERE` упрощается, в отличие от столбцов со строковыми данными постоянной длины, которые дополняются пробелами, а это требует дополнительных операций при выполнении сравнений с полями этого типа

Если код доступа к базе данных оформляется непосредственно на языке T-SQL, то в СУБД SQL Server автоматически разрешаются все проблемы, связанные со сравнением полей постоянной длины со строками, заполненными пробелами, и полей переменной длины. Иными словами, значение `'xx'`, помещенное в поле с типом данных `char(5)`, рассматривается как равное (при его сравнении) значению `'xx'`, записанному в поле типа `varchar(5)`. Тем не менее при использовании такого клиентского API-интерфейса, как ADO или ADO.NET, необходимые для такого сравнения преобразования не выполняются автоматически. При получении доступа к полю типа `char(5)` с помощью API-интерфейса ADO значение `'xx'` рассматривается как символы `xx` с тремя пробелами после них, поэтому результатом сравнения этого значения со строкой `'xx'` становится `False`. С другой стороны, если значение `'xx'` сохраняется в поле типа `varchar(5)`, то все конечные пробелы автоматически отсекаются, и поэтому сравнение этого значения со строковой константой `'xx'` с помощью средств ADO приводит к получению `True`.

Однако для хранения сокращенного обозначения отчества служащего, или так называемого среднего инициала, в большей степени подходит другой вариант. В действительности для представления инициала достаточно одной буквы, поэтому нет смысла отводить для соответствующего поля больший объем, а затем возвращать в систему неиспользуемое пространство. Кроме того, в данном случае строковый тип данных переменной длины фактически требует больше пространства, поскольку, например, тип `varchar` реализуется с применением не только того пространства памяти, в котором хранятся данные; для него также требуется немного дополнительного пространства, с помощью которого отслеживается длина хранимых данных. К тому же применительно к однобуквенному значению постоянной длины не возникает та проблема, о которой шла речь перед этим, касающаяся использования полей постоянной длины в операциях поиска, поскольку поле с инициалом может быть объявлено без лишнего пространства, дополняемого пробелами, поэтому достаточно лишь проверить, заданы ли в нем нужные данные.

По условиям рассматриваемой задачи имя и фамилия служащего не могут оставаться незаданными, поэтому в столбцах имени и фамилии не допускается наличие NULL-значений. А что касается среднего инициала, то соответствующее поле не имеет столь важного значения (и действительно, некоторые граждане США указывают свое имя и фамилию без отчества, т.е. вообще не имеют среднего инициала, тогда как жители Великобритании нередко имеют несколько средних инициалов), поэтому использование NULL-значений будет разрешено только для этого поля:

```
FirstName      varchar(25)          NOT NULL,
MiddleInitial  char(1)              NULL,
LastName       varchar(25)          NOT NULL,
```

Затем необходимо представить информацию о должности служащего. Ни один человек, работающий в компании, не может не иметь должности, поскольку иначе будет непонятно, за что он получает деньги, поэтому поле с обозначением должности также необходимо сделать обязательным:

```
Title          varchar(25)          NOT NULL,
```

К вопросу оплаты труда относится также информация о номере карточки социального обеспечения (или о подобном ему идентификационном номере, который принят в других странах, отличных от США), поскольку этот номер должен быть указан в налоговом отчете. В данном случае предусматривается использование типа данных `varchar` и допускается максимальное значение длины, равное 11 символам, поскольку идентификационные номера в разных странах имеют различную длину. С другой стороны, если известно, что в приложении достаточно предусмотреть возможность эксплуатации его в США, то, по-видимому, вполне допустимо ввести в определении этого поля обозначение типа `char(11)`:

```
SSN            varchar(11)          NOT NULL,
```

Затем необходимо определить, как осуществляется оплата труда служащих. На первый взгляд эта задача выглядит несложной, но иногда приходится учитывать дополнительные нюансы. Например, предположим, что регулярно происходит повышение заработной платы служащих. Если при этом необходимо вычислять новое значение по такому принципу, что берется предыдущая заработная плата и складывается с суммой последнего повышения, то возникает ситуация, в которой может использоваться вычисленный столбец, а новое значение зарплаты определяется как сумма предыдущей зарплаты и последнего повышения. Очевидно, что столбец `Salary` с данными о зарплате должен использоваться в запросах довольно часто, поэтому в действительности можно было бы задать на нем индекс, чтобы ускорить выполнение типовых запросов, но по различным причинам автор не хочет касаться в настоящей главе этой темы (сведения о том, к каким последствиям приводит применение индексов на вычисленных столбцах, изложены в главе 9), поэтому в качестве вычисленного столбца задан столбец с информацией о последнем повышении, `LastRaise`:

```
Salary         money                NOT NULL,
PriorSalary    money                NOT NULL,
LastRaise AS Salary - PriorSalary,
```

Отношения компании с каждым из ее служащих во многом зависят от того, каков стаж работы служащего в компании, поэтому необходимой является также информация о дате найма на работу, `HireDate`:

```
HireDate       smalldatetime        NOT NULL,
```

В этом определении заслуживает внимания то, что в целях экономии пространства для хранения даты используется тип данных `smalldatetime`, а не стандартный тип `datetime`. Безусловно, тип данных `datetime` обеспечивает хранение информации с точностью до долей секунды, а также охватывает гораздо более широкий диапазон дат по сравнению с другими типами. Но в данном случае нас главным образом интересует дата найма на работу, но никак не время, кроме того, в данном случае

приходится сталкиваться с ограниченным диапазоном календарных дат (поскольку ни один служащий не может, скажем, поступить на работу 50 лет тому назад и проработать еще 100 лет), поэтому тип данных smalldatetime не только соответствует потребностям в хранении информации, но и позволяет вдвое сократить объем используемого пространства.

> При выборе форматов полей для хранения значений даты и времени необходимо учитывать определенные компромиссы. С одной стороны, бывает очень приятно, если удается сэкономить пространство памяти и уменьшить объем данных, передаваемых по сети, с использованием типов данных, занимающих меньше места. С другой стороны, обнаруживается, что такой тип данных, как smalldatetime, является несовместимым с типами данных, применяемыми в некоторых других языках (включая Visual Basic). Тем не менее даже применение распространенного типа данных, подобного datetime, не позволяет получить гарантию, что указанная проблема никогда не возникнет, поскольку для некоторых моделей доступа к данным является более приемлемым подход, в котором даты передаются в виде данных типа varchar и обеспечивается неявное преобразование в значение поля datetime.

Дата окончания срока найма может оказаться неизвестной (а в отношении некоторых служащих хотелось бы даже надеяться, что они никогда не покинут свою компанию), поэтому должно быть разрешено использование в этом поле неопределенного значения:

```
    TerminationDate   smalldatetime          NULL,
```

Кроме того, в базе данных абсолютно необходимо иметь информацию о том, кто является руководителем и подчиненным (особенно в связи с тем, что иногда руководителям разрешается самим формировать свой кадровый состав!) и в каком отделе работает каждый служащий:

```
    ManagerEmpID      int                   NOT NULL,
    Department        varchar(25)           NOT NULL
)
```

Итак, для удобства дальнейшего изучения рассмотрим весь сценарий создания таблицы:

```
USE Accounting
CREATE TABLE Employees
(
    EmployeeID        int         IDENTITY    NOT NULL,
    FirstName         varchar(25)             NOT NULL,
    MiddleInitial     char(1)                 NULL,
    LastName          varchar(25)             NOT NULL,
    Title             varchar(25)             NOT NULL,
    SSN               varchar(11)             NOT NULL,
    Salary            money                   NOT NULL,
    PriorSalary       money                   NOT NULL,
    LastRaise AS Salary - PriorSalary,
    HireDate          smalldatetime           NOT NULL,
    TerminationDate   smalldatetime           NULL,
    ManagerEmpID      int                     NOT NULL,
    Department        varchar(25)             NOT NULL
```

Автор рекомендует после выполнения этого фрагмента кода снова вызвать процедуру sp_help применительно к рассматриваемой таблице, чтобы узнать, была ли она создана в соответствии с ожидаемым.

Оператор ALTER

Безусловно, разработчик после создания базы данных и определения в ней всех необходимых и очень удачных по своей структуре таблиц испытывает полное удовлетворение. Но жизнь не стоит на месте, поэтому иногда (а фактически гораздо чаще, чем хотелось бы) приходится выполнять распоряжения по изменению структуры таблицы, а не просто ее воссозданию. Кроме того, в процессе эксплуатации базы данных может потребоваться изменить размеры, местонахождение файлов или какие-то другие ее характеристики. Оператор ALTER позволяет осуществить все необходимые для этого действия.

Оператор ALTER во многом напоминает оператор CREATE, поэтому имеет в основном такую же синтаксическую структуру:

```
ALTER <object type> <object name>
```

Изучение этой структуры не составляет никакого труда, но более подробные сведения об операторе ALTER являются достаточно сложными. В этом можно будет сразу же убедиться, как только мы приступим к изучению указанного оператора, а после дальнейшего углубления в эту тему в следующей главе (когда речь пойдет об ограничениях) изложение станет действительно интересным (читай "сложным и запутанным"!).

Оператор ALTER DATABASE

Вначале рассмотрим, как можно модифицировать характеристики базы данных. В настоящем разделе рассматривается ряд примеров, позволяющих ознакомиться с тем, как осуществляются те или иные действия и какие для этого применяются синтаксические конструкции.

По-видимому, самым сложным аспектом применения оператора ALTER является определение того, какова ситуация, в которой должны осуществляться его действия. Для этой цели еще раз рассмотрим текущее состояние базы данных, как показано ниже.

```
EXEC sp_helpdb Accounting
```

Обратите внимание на то, что в этом случае, в отличие от предыдущего примера использования данной хранимой процедуры, для обозначения имени базы данных не применяются кавычки. Это связано с тем, что системная процедура sp_helpdb, как и многие другие процедуры такого типа, принимает в качестве параметра специальный тип данных, называемый sysname, а для этого типа данных кавычки являются необязательными, при условии, что процедуре передается действительное имя объекта, представленного в системе.

Так, полученные результаты должны полностью соответствовать тем, которые были получены сразу после создания базы данных (табл. 5.3 и 5.4).

Таблица 5.3. Данные о структуре таблицы Accounting (первый результирующий набор)

Name	db_size	owner	dbid	created	status	compatibility_level
Accounting	15.00 MB	sa	9	May 28 2000	Status=ONLINE, Updateability=READ_WRITE, UserAccess=MULTI_USER, Recovery=FULL, Version=598, Collation=SQL_Latin1_General_CP1_CI_AS, SQLSortOrder=52, IsAutoCreateStatistics, IsAutoUpdateStatistics, IsFullTextEnabled	90

Таблица 5.4. Данные о структуре таблицы Accounting (второй результирующий набор)

Name	fileid	filename	filegroup	size	maxsize	growth	usage
Accounting	1	c:\Program Files\Microsoft SQL Server\MSSQL.1\mssql\data\AccountingData.mdf	PRIMARY	10240 KB	51200 KB	5120 KB	data only
AccountingLog	2	c:\Program Files\Microsoft SQL Server\MSSQL.1\mssql\data\AccountingLog.ldf	NULL	5120 KB	25600 KB	5120 KB	log only

Допустим, что требуется внести некоторые изменения в характеристики базы данных. Например, предположим, что администратор базы данных получил распоряжение подготовить базу к импорту в нее большого объема данных. В настоящее время база данных имеет размер, равный всего лишь 15MB, а по нынешним меркам это не такой уж большой объем. Напомним, что параметру Autogrow присвоено значение on, поэтому можно было бы сразу же приступить к импорту, после чего СУБД SQL Server каждый раз автоматически наращивала бы размеры базы данных на 5MB по мере ее заполнения. Но следует помнить, что такие автоматические операции увеличения размеров базы данных фактически требуют от сервера выполнения довольно большого объема работы. Например, если должна быть произведена вставка данных в объеме 100MB, то серверу придется выполнять операции перераспределения базы данных по меньшей мере 16 раз (увеличить ее до 20MB, 25MB, 30MB и т.д.). А поскольку известно, что предстоит получение 100MB данных, то, по-видимому, более целесообразно заранее подготовиться к этому с помощью единственной операции. Для этого и используется команда ALTER DATABASE.

Команда ALTER DATABASE имеет следующий общий синтаксис:

```
ALTER DATABASE <database name>
  ADD FILE
      ([NAME = <'logical file name'>,]
       FILENAME = <'file name'>
       [, SIZE = <size in KB, MB, GB or TB>]
       [, MAXSIZE = < size in KB, MB, GB or TB >]
       [, FILEGROWTH = <No of KB, MB, GB or TB |percentage>]) [,...n]
```

```
          [ TO FILEGROUP filegroup_name]
[, OFFLINE ]
  |ADD LOG FILE
     ([NAME = <'logical file name'>,]
      FILENAME = <'file name'>
      [, SIZE = < size in KB, MB, GB or TB >]
      [, MAXSIZE = < size in KB, MB, GB or TB >]
      [, FILEGROWTH = <No KB, MB, GB or TB |percentage>])
  |REMOVE FILE <logical file name> [WITH DELETE]
  |ADD FILEGROUP <filegroup name>
  |REMOVE FILEGROUP <filegroup name>
  |MODIFY FILE <filespec>
  |MODIFY NAME = <new dbname>
  |MODIFY FILEGROUP <filegroup name> {<filegroup property>|NAME =
        <new filegroup name>}
  |SET <optionspec> [,...n ][WITH <termination>]
  |COLLATE <collation name>
```

Этот синтаксис является очень сложным, но в действительности некоторые разделы оператора ALTER DATABASE используются очень редко. Просто специалисты компании Microsoft постарались предусмотреть все возможные ситуации, в которых требуется применение этого оператора.

После краткого ознакомления со структурой оператора модификации базы данных остановимся на том, что просто увеличим размеры базы данных до 100MB:

```
ALTER DATABASE Accounting
  MODIFY FILE
  (NAME = Accounting,
   SIZE = 100MB)
```

Следует отметить, что при выполнении этого оператора, в отличие от оператора создания базы данных, не происходит вывод на внешнее устройство какой-либо информации о расположении базы данных; вместо этого формируется следующее краткое сообщение:

```
The command(s) completed successfully.
```

Очевидно, что с помощью этого сообщения нельзя узнать достаточно много о том, какой вид приобрела база данных, поэтому нам придется выполнить проверку самим:

```
EXEC sp_helpdb Accounting
```

Полученные результаты приведены в табл. 5.5 и 5.6.

Вполне очевидно, что попытка увеличения размера базы данных до 100MB завершилась успешно. Здесь заслуживает внимания то, что был превышен ранее установленный максимальный размер в 51200KB, но сообщение об ошибке не получено. Это связано с тем, что увеличение размера было выполнено в явной форме. Таким образом, можно сделать вывод, что в том же операторе модификации базы данных нужно было бы предусмотреть увеличение максимально допустимого размера базы данных. Ведь если мы остановимся на этом и просто позволим СУБД SQL Server наращивать размеры базы данных по мере необходимости, то попытка осуществления импорта окончится неудачей на полпути из-за того, что продолжают действовать ограничения по максимальному размеру. В этой ситуации необходимо также иметь в виду, что значение MAXSIZE увеличено лишь до нового явно заданного значения, поэтому возможность дальнейшего роста базы данных исключена.

Таблица 5.5. Данные о структуре таблицы Accounting (первый результирующий набор)

name	db_size	Owner	dbid	created	status	compatibility_level
Accounting	105.00 MB	Sa	9	May 28 2005	Status=ONLINE, Updateability=READ_WRITE, UserAccess=MULTI_USER, Recovery=FULL, Version=598, Collation=SQL_Latin1_General_CP1_CI_AS, SQLSortOrder=52, IsAutoCreateStatistics, IsAutoUpdateStatistics, IsFullTextEnabled	90

Таблица 5.6. Данные о структуре таблицы Accounting (второй результирующий набор)

name	fileid	Filename	filegroup	size	maxsize	growth	usage
Accounting	1	c:\Program Files\Microsoft SQL Server\MSSQL.1\mssql\data\AccountingData.mdf	PRIMARY	102400 KB	102400 KB	5120 KB	data only
AccountingLog	2	c:\Program Files\Microsoft SQL Server\MSSQL.1\mssql\data\AccountingLog.ldf	NULL	5120 KB	25600 KB	5120 KB	log only

По существу, все прочие наиболее часто выполняемые модификации на уровне базы данных осуществляются почти по такому же принципу. Тем не менее выбор всевозможных сочетаний параметров и опций оператора модификации базы данных является буквально бесконечным. Описание более сложных модификаций файловых групп и подобных изменений структуры базы данных выходят за рамки настоящей книги. Если же читателю требуется дополнительная информация по этой теме, то можно порекомендовать воспользоваться одной из книг, посвященных описанию процедур администрирования базы данных (а количество таких книг весьма велико).

Опции оператора модификации базы данных и спецификации завершения работы

В СУБД SQL Server предусмотрено несколько опций, которые могут быть заданы с помощью оператора ALTER DATABASE. К ним относятся, в частности, опции определения значений по умолчанию, касающихся конкретной базы данных, для большинства доступных опций SET (таких как ANSI_PADDING и ARITHABORT; подобной возможностью модификации базы данных удобно воспользоваться, если в приложении требуются индексированные или секционированные представления), опции состояния (например, определяющие эксплуатацию базы данных в однопользовательском режиме или режиме только чтения) и опции восстановления. Под воздействием различных опций SET функционирование базы данных изменяется, и в настоящей книге об этом часто идет речь при обсуждении соответствующих тем. А указанные новые функциональные возможности оператора модификации базы данных просто предо-

ставляют пользователю дополнительный способ корректировки значений параметров настройки, применяемых по умолчанию для любой конкретной базы данных.

В СУБД SQL Server предусмотрена также возможность управлять реализацией некоторых изменений, которые пользователь пытается внести в свою базу данных. Для осуществления многих модификаций требуется, чтобы пользователь получил исключительный контроль над базой данных, а этого трудно добиться, если в системе уже зарегистрированы другие пользователи. Поэтому СУБД SQL Server позволяет в корректной форме вынудить других пользователей завершить свои сеансы работы с базой данных, чтобы можно было провести намеченную модификацию базы данных. Применяемая при этом степень принуждения может изменяться, начиная с предоставления пользователям определенного времени в секундах (значение которого устанавливает разработчик, наметивший внесение изменений в базу данных) с последующей отменой регистрации других пользователей и заканчивая немедленным завершением всех транзакций (с их автоматическим откатом). Последний вариант предусматривает довольно непредвиденное (с точки зрения пользователя) завершение транзакций, а это — весьма радикальная мера, к которой нельзя прибегать без оглядки. Обычно на подобный шаг может пойти лишь администратор баз данных. Так или иначе, но дальнейшее обсуждение этой темы выходит за рамки настоящей книги.

Оператор ALTER TABLE

Гораздо более часто по сравнению с базой данных возникает необходимость модифицировать компоновку таблиц. Выполняемые при этом действия могут быть очень простыми (например, добавление нового столбца) или очень сложными (например, изменение типа данных).

Вначале рассмотрим основную синтаксическую структуру оператора модификации таблицы:

```
ALTER TABLE table_name
   {[ALTER COLUMN <column_name>
       { [<schema of new data type>].<new_data_type> [(precision [, scale])] max |
<xml schema collection>
       [COLLATE <collation_name>]
       [NULL|NOT NULL]
       |[{ADD|DROP} ROWGUIDCOL] | PERSISTED}]
    |ADD
       <column name> <data_type>
       [[DEFAULT <constant_expression>]
       |[IDENTITY [(<seed>, <increment>) [NOT FOR REPLICATION]]]]
       [ROWGUIDCOL]
       [COLLATE <collation_name>]
          [NULL|NOT NULL]
       [<column_constraints>]
       |[<column_name> AS <computed_column_expression>]
   |ADD
      [CONSTRAINT <constraint_name>]
      {[{PRIMARY KEY|UNIQUE}
         [CLUSTERED|NONCLUSTERED]
         {(<column_name>[ ,...n ])}
         [WITH FILLFACTOR = <fillfactor>]
         [ON {<filegroup> | DEFAULT}]
```

```
        |FOREIGN KEY
          [(<column_name>[ ,...n])]
          REFERENCES <referenced_table> [(<referenced_column>[ ,...n])]
          [ON DELETE {CASCADE|NO ACTION}]
          [ON UPDATE {CASCADE|NO ACTION}]
          [NOT FOR REPLICATION]
        |DEFAULT <constant_expression>
          [FOR <column_name>]
        |CHECK [NOT FOR REPLICATION]
          (<search_conditions>)
      [,...n][ ,...n]
        |[WITH CHECK|WITH NOCHECK]
    | { ENABLE | DISABLE } TRIGGER
      { ALL | <trigger name> [ ,...n ] }
  |DROP
    {[CONSTRAINT] <constraint_name>
      |COLUMN <column_name>}[ ,...n]
    |{CHECK|NOCHECK} CONSTRAINT
      {ALL|<constraint_name>[ ,...n]}
    |{ENABLE|DISABLE} TRIGGER
      {ALL|<trigger_name>[ ,...n]}
  | SWITCH [ PARTITION <source partition number expression> ]
      TO [ schema_name. ] target_table
      [ PARTITION <target partition number expression> ]
}
```

Очевидно, что оператор модификации таблицы не менее сложен, чем оператор создания таблицы.

Поэтому начнем изучение оператора модификации таблицы с того, что снова рассмотрим характеристики таблицы `Employees` базы данных `Accounting`:

```
EXEC sp_help Employees
```

В целях уменьшения объема представленной здесь информации полученные результаты отредактированы и показана лишь та часть, которая нас интересует, но фактически объем выводимой информации гораздо больше (табл. 5.7).

Таблица 5.7. Информация о структуре таблицы `Employees`

Column_name	Type	Computed	Length	Prec	Scale	Nullable
EmployeeID	int	no	4	10	0	no
FirstName	varchar	no	25			no
MiddleInitial	char	no	1			yes
LastName	varchar	no	25			no
Title	varchar	no	25			no
SSN	varchar	no	11			no
Salary	money	no	8	19	4	no
PriorSalary	money	no	8	19	4	no
LastRaise	money	yes	8	19	4	yes
HireDate	smalldatetime	no	4			no
TerminationDate	smalldatetime	no	4			yes
ManagerEmpID	int	no	4	10	0	no
Department	varchar	no	25			no

Предположим, что решено дополнительно предусмотреть хранение информации о предыдущем месте работы служащих (допустим, что это позволяет определить, какая компания может попытаться вернуть назад тех работников, которые проявили себя лучше всего). Для решения этой задачи достаточно ввести еще один столбец, а такая операция является довольно простой. Синтаксическая структура необходимого для этого оператора во многом напоминает структуру соответствующего оператора CREATE TABLE, не считая тех очевидных изменений, которые зависят от типа самого оператора:

```
ALTER TABLE Employees
  ADD
    PreviousEmployer  varchar(30)    NULL
```

Очевидно, что рассматриваемое задание оказалось несложным. Безусловно, можно было бы также ввести сразу несколько новых столбцов, если бы в этом возникла необходимость. Соответствующий оператор может выглядеть примерно таким образом:

```
ALTER TABLE Employees
  ADD
    DateOfBirth       datetime       NULL,
    LastRaiseDate     datetime       NOT NULL
      DEFAULT '2005-01-01'
```

Обратите внимание на то, что в последнем случае предусмотрено применение ключевого слова DEFAULT. Фактически использование этого ключевого слова еще подробно не рассматривалось (о нем речь пойдет в следующей главе), но автор в этом примере хотел бы отметить особый случай.

Если происходит добавление столбца с опцией NOT NULL в уже существующую таблицу, то возникает проблема – что делать со строками, которые уже содержат NULL-значения. В данном случае показано решение, состоящее в том, что предусмотрено значение, заданное по умолчанию. Это заданное по умолчанию значение затем используется для заполнения нового столбца в местах пересечения его со всеми строками, которые уже существуют в таблице.

Прежде чем завершить изучение данной темы, рассмотрим, к чему привело выполнение операции добавления столбца:

```
EXEC sp_help Employees
```

Информация об окончательно полученной структуре таблицы Employees приведена в табл. 5.8.

Вполне очевидно, что добавлены все необходимые столбцы. Но следует отметить, что добавленные столбцы оказались в конце списка столбцов. В СУБД SQL Server не предусмотрена возможность добавлять столбцы в любом конкретном месте. А если требуется поместить какой-то столбец в середину, то следует создать полностью новую таблицу (с другим именем), скопировать данные в новую таблицу, уничтожить существующую таблицу с помощью оператора DROP, а затем переименовать новую таблицу, назвав ее так же, как старую.

Таблица 5.8. Информация о структуре таблицы Employees

Column_name	Type	Computed	Length	Prec	Scale	Nullable
EmployeeID	int	no	4	10	0	no
FirstName	varchar	no	25			no
MiddleInitial	char	no	1			yes
LastName	varchar	no	25			no
Title	varchar	no	25			no
SSN	varchar	no	11			no
Salary	money	no	8	19	4	no
PriorSalary	money	no	8	19	4	no
LastRaise	money	yes	8	19	4	yes
HireDate	smalldatetime	no	4			no
TerminationDate	smalldatetime	no	4			yes
ManagerEmpID	int	no	4	10	0	no
Department	varchar	no	25			no
PreviousEmployer	varchar	no	30			yes
DateOfBirth	datetime	no	8			yes
LastRaiseDate	datetime	no	8			no

Указанная задача изменения порядка расположения столбцов может оказаться действительно очень сложной. Проблемы могут часто возникать даже при использовании некоторых инструментальных средств, предназначенных для автоматизации решения указанной задачи. Дело в том, что удаление текущей версии таблицы разрешается только после того, как будут уничтожены все ограничения внешнего ключа, которые ссылаются на данную таблицу. Это означает, что вначале необходимо уничтожить все внешние ключи, внести изменения, а затем снова ввести в действие внешние ключи. Тем не менее автоматическое удаление всех индексов, которые определены на старой таблице, при уничтожении существующей таблицы не происходит; это означает, что при создании новой версии таблицы необходимо учитывать, что в сценарии создания должно быть предусмотрено восстановление индексов, а это влечет за собой выполнение значительного объема дополнительной работы.

Но на этом описание всего, что нужно сделать в связи с введением модификаций в таблицу, не заканчивается. Хотя в настоящей книге тематика представлений фактически еще не рассматривалась, автор считает себя обязанным указать предварительно даже в этом разделе, что происходит с представлениями после добавления столбца. Следует знать, что новый столбец не появляется в представлении до тех пор, пока представление не будет уничтожено и вновь создано, даже если базовым оператором при создании представления служит оператор `SELECT *`*. Дело в том, что в целях повышения производительности задача разрешения имен столбцов в представлениях выполняется только во время создания представления. Это означает, что во всех представлениях, которые уже были созданы ко времени добавления новых столбцов, задача разрешения имен столбцов уже выполнена с использованием существовавшего ранее списка столбцов, поэтому необходимо либо уничтожить и вновь создать каждое такое представление, либо воспользоваться оператором* `ALTER VIEW`*, чтобы обновить каждое представление.*

Оператор DROP

Выполнение оператора DROP приводит к удалению любого указанного в нем объекта (объектов). Оператор DROP является весьма быстродействующим и удобным, а синтаксис этого оператора остается полностью одинаковым применительно ко всем основным объектам СУБД SQL Server (таблицам, представлениям, хранимым процедурам, триггерам и т.д.). Этот синтаксис выглядит примерно таким образом:

```
DROP <object type> <object name> [, ...n]
```

Фактически оператор удаления — один из самых простых операторов SQL. При желании с помощью этого оператора мы можем одновременно удалить обе рассматриваемые таблицы:

```
USE Accounting

DROP TABLE Customers, Employees
```

Выполнение этого оператора заканчивается успешно.

> При использовании оператора удаления необходимо соблюдать исключительную осторожность, особенно в связи с тем, что перед его выполнением от пользователя не требуется подтверждение того, что рассматриваемый объект действительно должен быть удален. СУБД SQL Server просто действует на основании предположения, что решение пользователя обосновано, и сразу же удаляет указанный объект (объекты).

Еще раз напомним приведенную в начале данной главы рекомендацию, касающуюся введения оператора USE в начало каждого сценария. Рассматриваемый пример показывает, почему соблюдение этой рекомендации является столь важным. Дело в том, что таблицы Customers и Employees имеются также и в базе данных Northwind, а в настоящее время у нас нет намерения их удалять. Разумеется, по стечению обстоятельств попытка выполнить данный оператор не привела бы к непреднамеренному удалению указанных таблиц из базы данных Northwind из-за некоторых проблем, связанных с неправильным выбором последовательности удаления, но удалить таблицу [Order Details] или, скажем, Shipments ничего бы не помешало.

Синтаксическая структура оператора удаления всей базы данных остается в основном той же самой. Теперь приступим к удалению базы данных Accounting:

```
USE master

DROP DATABASE Accounting
```

После выполнения этого оператора в области окна Results должно появиться следующее:

```
Deleting database file 'c:\Program Files\Microsoft SQL Server\mssql\data\
AccountingLog.ldf'.
Deleting database file 'c:\Program Files\Microsoft SQL Server\mssql\data\
AccountingData.mdf'.
```

В ходе этого может возникнуть такая ситуация, что появится сообщение об ошибке, которое указывает, что база данных не может быть удалена, поскольку находится

в использовании. В таком случае можно выполнить проверку ряда перечисленных ниже условий.

- Убедитесь в том, что база данных, которую вы пытаетесь удалить, не определена как текущая в программе Management Studio (иными словами, убедитесь в том, что вы не работаете в текущий момент с той базой данных, которую пытаетесь удалить).
- Проверьте, нет ли каких-либо других открытых соединений (с помощью программы Management Studio или процедуры `sp_who`), для которых база данных, предназначенная для удаления, является текущей базой данных.

Автор обычно предотвращает возможность возникновения первой из указанных ситуаций так, как показано в приведенном примере кода, т.е. переключается на использование базы данных `master`. А отсутствие второй ситуации приходится проверять вручную; автор обычно полностью закрывает все прочие сеансы, чтобы быть уверенным в том, что эта ситуация не имеет места.

Использование инструментальных средств с графическим интерфейсом пользователя

В настоящей главе уже уделено достаточно внимания изучению синтаксиса операторов, вполне обеспечивающих создание базы данных и целого ряда таблиц. Поэтому на время сменим тему и перейдем к рассмотрению графических инструментальных средств программы Management Studio, которые позволяют создавать таблицы и устанавливать между ними связи. Начиная с этого раздела, в настоящей книге будет идти речь не только о том, как подготовить нужный код, но и об использовании инструментальных средств для создания основного объема необходимого кода.

Создание базы данных с помощью программы Management Studio

После вызова на выполнение программы SQL Server Management Studio и развертывания узла `Databases` откроется окно, которое выглядит так, как показано на рис. 5.1.

Этот рисунок представляет собой копию экрана, который показывает, что в окне на компьютере автора все еще изображено обозначение базы данных `Accounting`*, даже несмотря на то, что в предыдущем примере эта база данных была удалена. А что касается читателя, то он в своей программе может обнаружить или не обнаружить обозначение этой базы данных, в зависимости от того, была ли программа Management Studio открыта ко времени удаления базы данных или она была открыта после того, как произошло удаление базы данных в программе Query Analyzer.*

Указанное различие между изображениями обусловлено тем, что в предыдущих версиях СУБД SQL Server те инструментальные средства, которые в настоящее время входят в состав программы Management Studio, обеспечивали регулярное обновление информации об имеющихся базах данных. А в текущей версии обновление происходит, только если обнаруживается причина этого обновления (например, если какой-то объект удален с помощью программы Management Studio Object Explorer, а не путем выполнения оператора в окне `Query` *или, возможно, явно выбрана команда обновления изображения). Причиной перехода к такой организации работы*

стала потребность в повышении производительности. Общепризнанно то, что старая версия 6.5 программы Enterprise Manager отличалась низкой производительностью, поскольку в ней постоянно осуществлялся обмен пакетами в целях "опроса" сервера. В связи с внедрением указанного нового подхода производительность существенно возросла, но в связи с этим отображаемая в программе информация не всегда является наиболее актуальной.

Подводя итог сказанному, следует отметить, что, обнаружив в программе Management Studio то, чего не следовало быть, и не обнаружив то, что должно быть, попытайтесь нажать клавишу <F5> (связанную с командой обновления), после чего на экране должно появиться обновленное изображение.

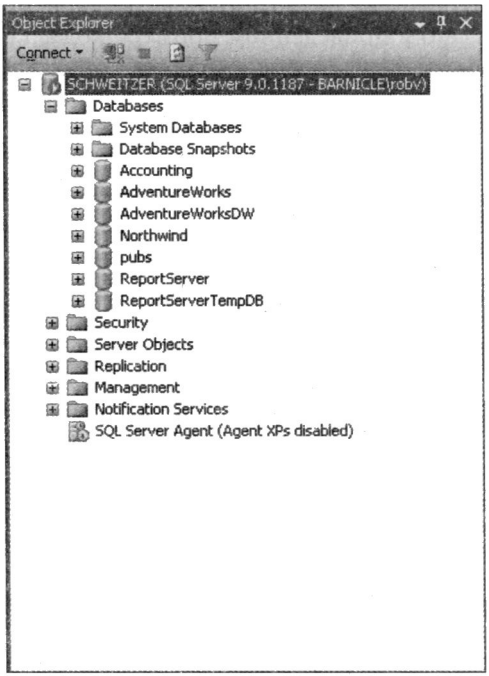


Рис. 5.1. Окно Object Explorer

А теперь попытайтесь щелкнуть правой кнопкой мыши на узле Databases и выберите опцию New Database... .

В результате этого откроется диалоговое окно Database Properties, которое позволяет ввести информацию о том, какие свойства должна иметь база данных, намеченная к созданию. Выберем такие же характеристики, которые использовались при создании базы данных `Accounting` в начале настоящей главы. Прежде всего необходимо задать имя самой базы данных и ввести информацию о ее размерах (рис. 5.2).

Описание всей этой вкладки указанного диалогового окна дается впервые, поэтому рассмотрим, что имеется на этой вкладке.

Прежде всего, здесь предусмотрено поле для указания имени базы данных. Заполнение этого поля является весьма несложной задачей. Перед этим уже была успешно создана база данных `Accounting`, а поскольку она уже уничтожена, то нет причин, по которым нельзя было бы присвоить имя `Accounting` новой базе данных.

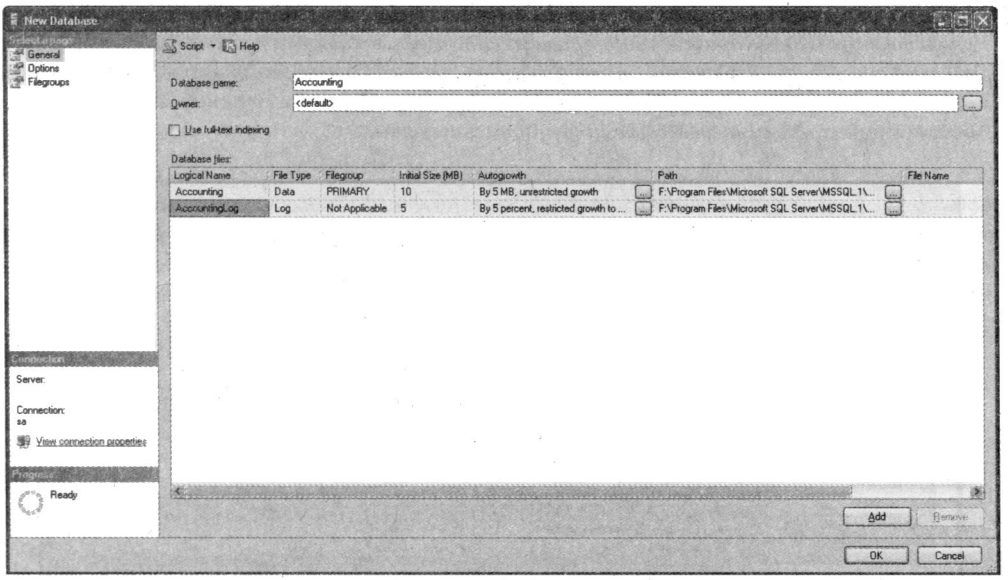


Рис. 5.2. Информация о размерах базы данных (вкладка `General`)

Затем необходимо ввести информацию об имени файла, размере и параметрах роста.

Автор развернул это диалоговое окно вручную, чтобы можно было показать его на рисунке полностью. Но фактически может оказаться, что на экране отображается не вся эта информация, поскольку применяемый по умолчанию размер диалогового окна немного меньше, чем требуется для изображения всей информации. Тем не менее достаточно захватить курсором угол диалогового окна и растянуть его до нужных размеров, чтобы можно было видеть всю дополнительную информацию

Теперь перейдем к вкладке Options, на которой имеется большое количество дополнительных параметров настройки (рис. 5.3).

По-видимому, в данном окне наибольшего внимания заслуживает параметр с обозначением способа упорядочения (`Collation`). Начиная с версии SQL Server 2000, появилась возможность предусматривать применяемый способ упорядочения отдельно для каждой базы данных (а фактически, при желании, — отдельно для каждого столбца). Тем не менее в подавляющем большинстве инсталляций не вводятся изменения, отличающиеся от тех значений, которые предлагаются для использования по умолчанию во время инсталляции сервера (и действительно, в этом нет особой необходимости, ведь все параметры уже были тщательно продуманы заранее). Однако теперь имеется возможность вносить изменения, распространяющиеся только на ту базу данных, для которой это необходимо.

Выбор того или иного способа упорядочения иногда становится необходимым не только в связи с переходом к использованию кодовой страницы, отличной от применяемой для англоязычного текста. Дело в том, что даже при упорядочении строковых данных, представленных на английском языке, приходится учитывать, что в некоторых приложениях не проводится различие между прописными и строчными буквами, тогда как другие приложения являются чувствительными к регистру. Разумеется, при использовании прежних версий приходилось

разворачивать несколько серверов, чтобы иметь возможность применять различные способы упорядочения. Кроме приведенного примера ситуации, в которой приходится применять несколько способов упорядочения, можно указать необходимость учитывать различия между диалектами, с которыми приходится сталкиваться во многих странах мира, даже если в этих странах применяется единый национальный язык.

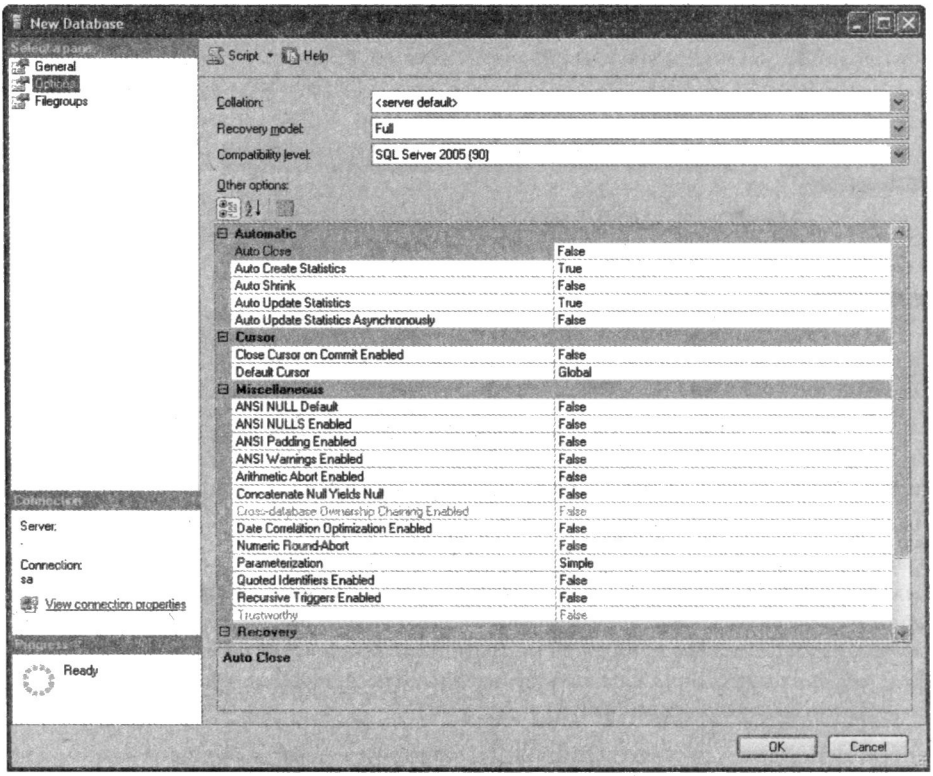


Рис. 5.3. Вкладка Options

Следующим важным параметром является уровень совместимости (compatibility level). От этого параметра зависит то, будут ли поддерживаться некоторые синтаксические конструкции и ключевые слова, предусмотренные в версии SQL Server 2005. Вполне очевидно, что с помощью этого параметра настройки можно обеспечить возврат к использованию тех ключевых слов и функциональных средств, которые в большей степени соответствуют предыдущим версиям, если в этом возникнет необходимость при разработке какого-то конкретного приложения.

Выбор остальных параметров настройки зависит от требований к конкретной инсталляции, но в основном все эти параметры действуют так, как было описано выше в данной главе.

Итак, введите в качестве всех параметров настройки в основном такие же значения, которые были показаны в предыдущих примерах данной главы, чтобы проверить, как они применяются, и щелкните на кнопке OK. Для фактического создания базы данных потребуется определенное время, после чего вы обнаружите, что обозначение новой базы данных добавлено к дереву объектов.

Теперь разверните дерево объектов, чтобы ознакомиться с различными элементами, находящимися под узлом Accounting, и выберите узел Database Diagrams. Щелкните правой кнопкой мыши на этом узле, чтобы открыть диалоговое окно с указанием, что в базе данных отсутствуют некоторые объекты, необходимые для обеспечения работы средств создания диаграммы структуры объектов базы данных (рис. 5.4). Щелкните на кнопке Yes.

Следует отметить, что указанное сообщение должно появиться только при первой попытке создания диаграммы объектов для рассматриваемой базы данных. В СУБД SQL Server диаграммы отслеживаются в специальных таблицах, создаваемых в базе данных только в тот момент, когда пользователь выражает желание ознакомиться с диаграммой, в которой используются такие специальные таблицы.

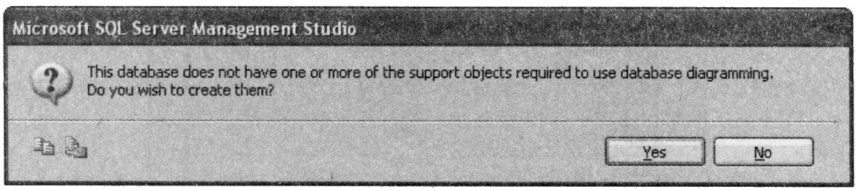


Рис. 5.4. Запрос пользователю, связанный с необходимостью создания дополнительных объектов в базе данных

После этого откроется диалоговое окно Add Table (рис. 5.5), которое позволяет определить, какие таблицы должны быть включены в определенную диаграмму; при желании можно создать несколько диаграмм и отобразить в каждой из них отдельный подраздел (или субмодель) общей схемы базы данных. А в данном случае отображается только одна таблица. (Напомним, что таблицы `Customers` и `Orders` были удалены некоторое время тому назад и осталась пустая база данных, в том смысле, что в ней нет таблиц, созданных пользователем для своих целей.)

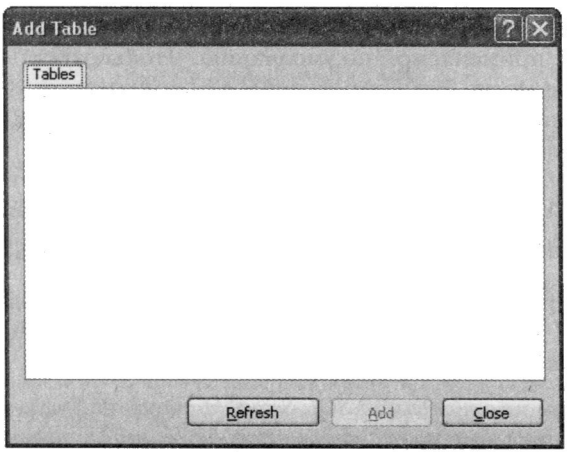


Рис. 5.5. Диалоговое окно Add Table

На данном этапе достаточно щелкнуть на кнопке Cancel, чтобы открыть пустое окно диаграммы. Удобным свойством этого окна является то, что в нем можно ввести таблицу, либо щелкнув правой кнопкой мыши и выбрав соответствующую опцию, либо щелкнув левой кнопкой на пиктограмме New table панели инструментов. После выбора новой таблицы СУБД SQL Server выводит приглашение для ввода имени, которое должно быть присвоено новой таблице. После этого отображается довольно удобное диалоговое окно, позволяющее последовательно определять столбцы таблиц, руководствуясь удобными надписями, которые подсказывают, чем должна быть заполнена та или иная часть окна (рис. 5.6).

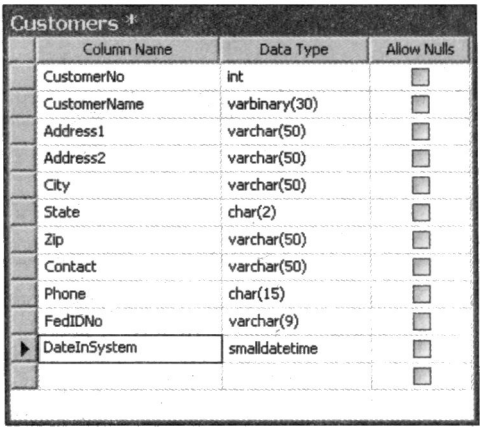


Рис. 5.6. Окно с вводимым определением таблицы

Автор заранее выполнил всю необходимую работу и ввел информацию о столбцах в полном соответствии со структурой рассматривавшейся в предыдущих примерах таблицы `Customers`, но в данном случае необходимо также определить, что первый столбец представляет собой столбец идентификации. К сожалению, создается впечатление, что нет ни одного способа решить указанную задачу с помощью представленной здесь сетки, применяемой по умолчанию. Чтобы откорректировать перечень элементов, которые могут быть определены для конкретной таблицы, необходимо щелкнуть правой кнопкой мыши в диалоговом окне редактирования и выбрать команду Table View⇨Modify Custom.

После этого отобразится список элементов (рис. 5.7), из которого мы можем выбрать необходимый элемент. На данный момент достаточно выбрать тот дополнительный элемент, который нам требуется, `Identity`, и связанные с ним элементы `Seed` и `Increment`.

Теперь снова вернемся к диалоговому окну редактирования и выберем команду Table View⇨Custom для просмотра столбца идентификации (рис. 5.8), после чего можем приступить к заполнению определения таблицы.

Следует отметить, что программа SQL Server Management Studio в данном случае выполняет действия, которые могут оказаться неожиданными для пользователя. Если в указанном окне не будет установлен флажок, в соответствии с которым вид таблицы, определяемый пользователем, должен стать применяемым по умолчанию, то программа Management Studio сохранит информацию о том, какой вид отображения таблицы (Table View) выбран пользователем, но не активизирует данный конкретный выбранный пользователем вид.

В результате этого пользователь не обнаружит внесенных им изменений после выхода из указанного диалогового окна. Поэтому еще раз отметим, что после изменения выбранного вида таблицы щелкните в диалоговом окне редактирования правой кнопкой мыши и снова выберите команду Table View⇨Custom. *После этого оно должно принять такой вид, как показано на рис. 5.8.*

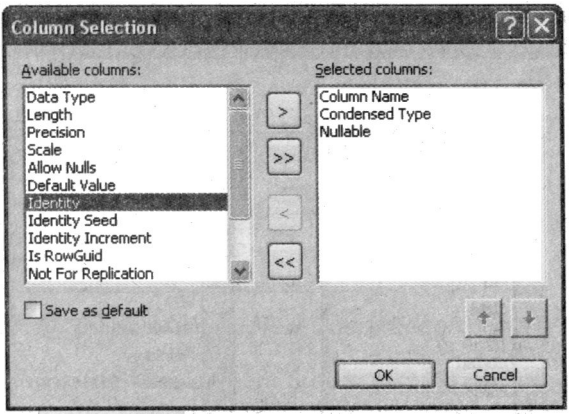


Рис. 5.7. Окно *Column Selection*

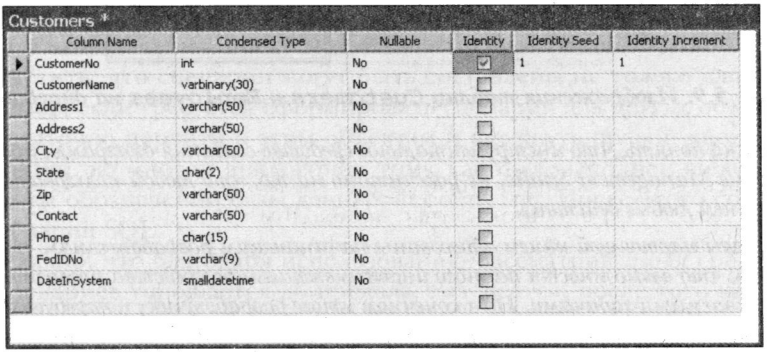


Рис. 5.8. Окно определения таблицы, в котором показаны элементы *Identity, Identity Seed и Identity Increment*

После того как опции определения таблицы будут заполнены, можно сохранить изменения, после чего таблица создается от имени пользователя.

Безусловно, выбор способа отображения – дело личного вкуса, но сам автор предпочитает на данном этапе иметь визуальный контроль над всей информацией, вплоть до имен столбцов. Этого можно добиться, щелкнув на пиктограмме Show *панели инструментов, но автор предпочитает щелкнуть правой кнопкой мыши на обозначении таблицы и выбрать команду* Table View⇨Column Names. *По мнению автора, благодаря этому экономится значительный объем пространства экрана и остается больше места для работы над остальными таблицами.*

Теперь читатель должен попытаться создать таблицу `Employees` в том виде, в каком она была определена выше в данной главе. Осуществляемые при этом действия

должны быть в основном такими же, как и при создании таблицы Customers, за исключением одного небольшого нюанса, который связан с наличием вычисленного столбца. Чтобы определить вычисленный столбец, еще раз выберите команду Modify Custom (из меню, которое всплывает в окне после щелчка правой кнопкой мыши) и добавьте столбец с формулой. После этого достаточно ввести нужную формулу (в данном случае Salary-PriorSalary). После ввода сведений обо всех столбцах сохраните определение новой таблицы (дайте положительный ответ на вопрос о ее сохранении в диалоговом окне подтверждения), после чего на диаграмме должны появиться две таблицы (рис. 5.9).

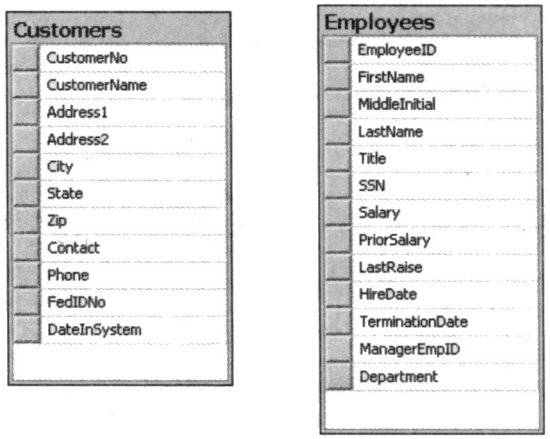


Рис. 5.9. Изображения таблиц Customers и Employees на диаграмме

Очень важно понять, что инструментальное средство создания диаграмм, предусмотренное в программе Management Studio, не рассчитано на то, что любой пользователь может выполнять в нем любые действия.

Первые главы настоящей книги адресованы начинающим разработчикам, поэтому можно допустить, что возможности данного инструментального средства позволят справиться с рассматриваемыми задачами. Но в конечном итоге разработчику потребуются более усовершенствованные (и вместе с тем гораздо более дорогостоящие) инструментальные средства, позволяющие ему справиться со сложными задачами проектирования базы данных.

Основные сведения о создании сценариев с помощью программы Management Studio

В завершении данной главы рассмотрим основные сведения о том, как воспользоваться программой Management Studio для написания сценариев. На этом этапе будет дано лишь краткое введение в указанную тему, а в дальнейшем, после изучения многих других объектов, поддерживаемых инструментальным средством обработки сценариев, перейдем к более подробному описанию.

Для создания сценариев необходимо вызвать на выполнение программу Management Studio и щелкнуть правой кнопкой мыши на обозначении базы данных, для которой требуется создать сценарии (в данном случае будет показано, как создать сценарии для базы данных Accounting). Во всплывающем меню выберите команду Script Database As⇨CREATE TO⇨New Query Editor Window (рис. 5.10).

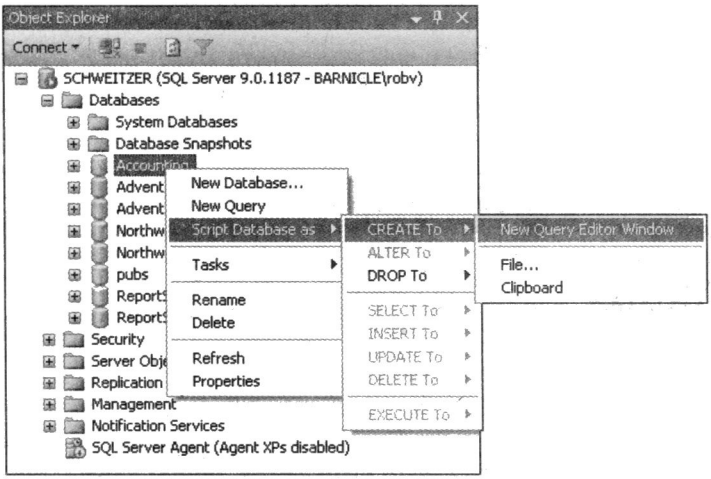


Рис. 5.10. Вызов команды *New Query Editor Window*

Очевидно, что в программе Management Studio вырабатывается намного больше кода по сравнению с тем, что было введено нами при создании базы данных. Но в этом нет ничего необычного. Дело в том, что программа при составлении сценариев явно показывает все необходимые параметры настройки базы данных, а мы при составлении сценария в основном исходили из того, что вместо незаданных значений будут использоваться значения, предусмотренные по умолчанию.

Следует отметить, что сценарии могут быть составлены не только для всей базы данных. Например, если требуется сформировать сценарии для других объектов базы данных, то достаточно перейти к этим объектам и щелкнуть правой кнопкой мыши на их обозначении, во многом аналогично тому, как был выполнен щелчок правой кнопкой мыши на обозначении базы данных `Accounting`. После этого создается необходимый сценарий SQL.

Вполне очевидно, что благодаря использованию рассматриваемых средств задача составления сценариев упрощается до предела. Разумеется, после того как база данных станет достаточно сложной, немного усложнится и сам способ формирования относящихся к ней сценариев по сравнению с данным конкретным примером, но все равно этот процесс остается гораздо менее сложным по сравнению с написанием всех необходимых сценариев вручную. В действительности читателю станет гораздо проще работать только после того, как он изучит все возможности формирования сценариев, а также освоит намного больше информации по данной теме, изучив материал остальных глав настоящей книги.

Резюме

В настоящей главе приведены основные сведения об использовании операторов `CREATE`, `ALTER` и `DROP` для создания, модификации и удаления баз данных и таблиц. Безусловно, на этом описание указанных операторов далеко не исчерпывается, поэтому им будет посвящена значительная часть изложения в остальных главах. Кроме того, рассмотрен целый ряд опций, применяемых в определениях баз данных и таблицах, которые позволяют получить полный контроль над данными. Наконец, в

этой главе приведены вводные сведения об использовании программы Management Studio для упрощения задач администрирования данных, а также для более простого проектирования структуры объектов и создания сценариев.

На данный момент читатель уже имеет достаточную подготовку для того, чтобы приступить к изучению таких сложных тем, как компоновка таблиц, к освоению понятий нормализации и переходу к созданию более общего проекта базы данных. Тем не менее, прежде чем перейти к рассмотрению указанных вопросов проектирования, необходимо ознакомиться с информацией об ограничениях и ключах.

Упражнения

5.1. Создайте сценарии SQL для таблиц `Customers` и `Employees`, используя средства формирования сценариев программы Management Studio.

5.2. Подготовьте сценарий создания базы данных `MyDB` без использования программы Management Studio. Установите начальный размер базы данных `17MB` и журнала `5MB`; увеличение размеров базы данных и журнала должно происходить с шагом `5MB`.

5.3. Создайте таблицу `Foo` с единственным символьным полем переменной длины `Col1`; установите ограниченный размер столбца `Col1`, равный 50 символам.

6

Ограничения

В предыдущих главах речь уже шла об ограничениях, а данная глава полностью посвящена этой теме. В последних нескольких версиях SQL Server были внесены значительные изменения, касающиеся поддержки ограничений, и эта тенденция продолжилась с выпуском SQL Server 2005.

До сих пор в настоящей книге тема ограничений неоднократно затрагивалась в различных контекстах, но мы обязаны снова вернуться к изложенному материалу, чтобы читателю было легче приступить к изучению данной темы, если он решил сразу же перейти к текущей главе.

> Ограничение — это, прежде всего, формулировка требований к данным. Ограничения устанавливаются на уровне столбца или таблицы и гарантируют соответствие данных определенным правилам обеспечения целостности данных.

В связи с этим следует еще раз подчеркнуть мысль о том, что за обеспечение целостности данных отвечают не программы, в которых используется база данных, а непосредственно база данных, о чем уже было сказано в главах 1 и 2. И даже краткие размышления на эту тему позволяют прийти к выводу, что такие свойства базы данных действительно являются очень удобными. Дело в том, что команды на вставку, обновление и удаление данных поступают в базу данных из многих источников. Причем даже в автономных приложениях (в этом контексте таковыми считаются приложения, в которых доступ к базе данных осуществляет только одна программа) обращения к одной и той же таблице могут поступать из различных мест программы. Но в действительности чаще всего возникают еще более сложные ситуации. Например, одновременно с доступом со стороны приложения иногда изменения в данные вносит администратор базы данных (или сам программист, если он совмещает функции администратора базы данных) в целях устранения обнаруженных проблем. А на практике нередко возникают настолько сложные ситуации, что к одному-един-

ственному фрагменту данных, не говоря уже обо всей базе данных, прокладываются буквально сотни различных путей доступа для обеспечения работы сложного многопользовательского приложения.

Возможность возложить ответственность за поддержание целостности данных на саму базу данных появилась не сразу, но благодаря внедрению связанных с этим средств произошли буквально революционные изменения в методах управления базой данных. Это отнюдь не означает, что любая попытка вставки данных в базу данных должна теперь осуществляться успешно, но в ответ на возможные нарушения в работе, которые могут при этом возникнуть, современная база данных действует превентивно, а не пытается лишь исправить возникающие ошибки. В настоящее время существует возможность заблаговременно, еще на этапе разработки базы данных, исключать значительную часть причин появления ошибочных данных в базе данных, связанных с неправильной работой приложения, поскольку для обеспечения целостности данных может применяться сама база данных. Для этого в основном служат ограничения (кроме них, к числу наиболее широко применяемых средств обеспечения целостности данных относятся типы данных и триггеры).

В настоящей главе приведен общий обзор следующих трех типов ограничений.

- Ограничения сущностей.
- Ограничения домена.
- Ограничения ссылочной целостности.

Кроме того, более подробно будут рассматриваться конкретные методы реализации каждого из ограничений этих типов, включая перечисленные ниже.

- Ограничения первичного ключа (`PRIMARY KEY`).
- Ограничения внешнего ключа (`FOREIGN KEY`).
- Ограничения уникальности (`UNIQUE`, именуемые также ограничениями альтернативного ключа).
- Ограничения проверки (`CHECK`).
- Ограничения заданных по умолчанию значений (`DEFAULT`).
- Правила.
- Заданные по умолчанию значения (подобные ограничениям `DEFAULT`, но все же отличные от них).

> Два из наиболее широко используемых видов действий по обеспечению ссылочной целостности — каскадные обновления и каскадные удаления — впервые получили свою поддержку в версии SQL Server 2000. До тех пор пользователи выражали свое недовольство отсутствием поддержки этих средств. Но компания Microsoft оставила нереализованными некоторые другие области обеспечения ссылочной целостности по стандарту ANSI; эти средства были введены в версии SQL Server 2005. Каскадные действия и другие средства обеспечения ссылочной целостности, соответствующие стандарту ANSI, будут подробно описаны при рассмотрении ограничений внешнего ключа, `FOREIGN KEY`.

Кроме того, в настоящей главе будут очень кратко описаны триггеры и хранимые процедуры как методы реализации правил обеспечения целостности данных (гораздо больший объем сведений по этим темам будет приведен в следующих главах).

Типы ограничений

Предусмотрен целый ряд различных способов реализации ограничений, но все эти способы относятся к одной из трех категорий – ограничения сущностей, ограничения домена и ограничения ссылочной целостности (рис. 6.1).

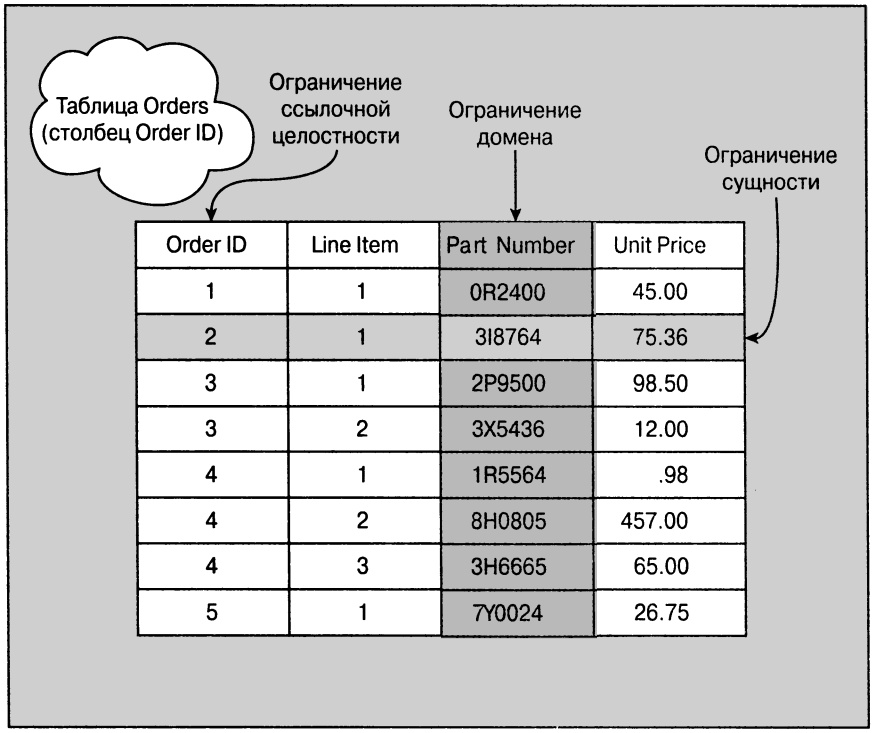


Рис. 6.1. Основные типы ограничений

Ограничения домена

Ограничения домена распространяются на один или несколько столбцов. Под этими ограничениями подразумеваются способы обеспечения того, чтобы какой-то конкретный столбец или ряд столбцов соответствовал определенным критериям. Эти ограничения применяются при вставке или обновлении строки без учета того, что в таблице имеются какие-либо иные другие строки; интерес представляют только данные рассматриваемого столбца.

Например, как ограничение домена рассматривается такое ограничение, с помощью которого требуется обеспечить наличие в столбце `UnitPrice` только значений, больших или равных нулю. С учетом этого ограничения будет отвергнута любая строка, в которой значение `UnitPrice` не отвечает указанному условию, но тем самым фактически принудительно поддерживается правило обеспечения целостности, позволяющее гарантировать соответствие указанному ограничению всего

столбца (независимо от того, сколько строк имеется в рассматриваемой таблице). В качестве домена служит столбец, а ограничение представляет собой ограничение домена.

Ограничения указанного типа будут рассматриваться при описании ограничений CHECK, правил, заданных по умолчанию значений и ограничений DEFAULT.

Ограничения сущности

Ограничения сущности полностью относятся к отдельным строкам. В действительности в ограничении этого типа не рассматривается весь столбец как таковой; интерес представляет только какая-то конкретная строка. Самым наглядным ограничением этого типа является такое ограничение, согласно которому в каждой строке таблицы должно присутствовать уникальное значение одного столбца или комбинации столбцов.

На первый взгляд может показаться, что такое определение ограничения сущности, согласно которому, допустим, какой-то столбец должен содержать уникальные значения, полностью совпадает с определением ограничения домена. Но фактически дело обстоит иначе.

В формулировке ограничения сущности ничего не сказано о том, что данные столбца должны соответствовать какому-то определенному формату или что значения в столбце должны быть больше или меньше какой-то величины. Единственное, что сказано, относится только к определенной строке и предъявляемое к ней требование состоит лишь в том, чтобы содержащееся в ней значение больше не встречалось в какой-либо другой строке в той же таблице.

Ограничения такого рода будут рассматриваться в контексте описания ограничений PRIMARY KEY и UNIQUE.

Ограничения ссылочной целостности

Ограничения ссылочной целостности создаются в том случае, если значения в одном столбце должны согласовываться со значениями в другом столбце (либо в той же таблице, либо, гораздо чаще, в другой таблице).

Предположим, что разрабатывается приложение, предназначенное для приема заказов на товары, и что это приложение должно обеспечивать получение оплаты с помощью кредитных карточек. А чтобы иметь возможность получения платежей от компании — эмитента кредитных карточек, необходимо заключить с этой компанией торговое соглашение в определенной форме. Кроме того, требуется, чтобы служащие, принимающие заказы, не брали для оплаты кредитные карточки, выпущенные компаниями, от которых не обеспечивается получение платежей. Именно в таком случае вступают в силу ограничения ссылочной целостности. С помощью этого ограничения может быть создана таблица, которая в дальнейшем именуется таблицей домена или поисковой таблицей. Таблицей домена называется таблица, единственным назначением которой является предоставление ограниченного списка допустимых значений. В данном случае может быть создана таблица, которая выглядит так, как показано в табл. 6.1.

Таблица 6.1. Пример таблицы домена

Поле CreditCardID	Поле CreditCard
1	VISA
2	MasterCard
3	Discover Card
4	American Express

После этого появляется возможность создать одну или несколько таблиц, которые ссылаются на столбец CreditCardID таблицы домена. Чтобы иметь возможность использовать средства обеспечения ссылочной целостности, необходимо предусматривать в любой таблице (такой как таблица с заказами, Orders), которая определена как ссылающаяся на таблицу CreditCard, такой столбец, который согласуется со столбцом CreditCardID таблицы CreditCard. С этого момента в каждой строке, вставляемой в таблицу, ссылающуюся на ограничение, должно присутствовать только такое значение, которое имеется в заданном списке домена (иными словами, этому значению должна соответствовать одна из строк в таблице CreditCard).

Дополнительная информация по этой теме будет приведена при описании ограничений FOREIGN KEY ниже в данной главе.

Способы именования ограничений

Прежде чем перейти к рассмотрению различных нюансов использования ограничений, сделаем небольшое отступление и рассмотрим проблему выбора способа именования ограничений.

> Все возможные виды ограничений, рассматриваемые в настоящей главе, должны быть обозначены именем, но разработчик не обязан сам задавать такое имя. Иными словами, всегда можно воспользоваться тем, что СУБД SQL Server предоставляет имя для того ограничения, для которого имя не было предусмотрено разработчиком. Тем не менее следует избегать соблазна воспользоваться такой возможностью, поскольку вскоре обнаруживается, что имена, создаваемые СУБД SQL Server, не вполне приемлемы.

В качестве примера можно указать, что имена, формируемые системой, принимают такой вид: PK__Employees__145C0A3F. В данном случае демонстрируется имя, сформированное СУБД SQL Server для первичного ключа таблицы Employees базы данных Accounting. Информация о том, как создается первичный ключ, будет приведена ниже в данной главе, но в этом примере имени, сформированного системой, PK сокращенно обозначает первичный ключ, primary key (и это — один из компонентов имени ограничения, который следует предусматривать в любом случае), часть имени ограничения Employees указывает на таблицу Employees, к которой относится это ограничение, а остальная часть имени — это значение, сформированное случайным образом для обеспечения уникальности ограничения. Очевидно, что имя, сформированное системой, не подходит для повседневного использования, поэтому для разработчика имеет смысл применять такой способ именования, только если он создает первичные ключи с помощью сценария. А в том случае, если бы для создания этой

таблицы использовалась программа Management Studio, то ограничение получило бы имя PK_Employees.

Но основной недостаток имен, сформированных системой, состоит не в их сложности, а в том, что эти имена не раскрывают сути применяемых ограничений; например, как показано ниже в данной главе, при использовании ограничения CHECK системой формируется имя, напоминающее нечто вроде CK__Customers__22AA2996. По этому имени можно определить, что оно относится к ограничению CHECK, однако невозможно что-либо узнать, в чем состоит характер соответствующей проверки CHECK.

Учитывая то, что на одной таблице может быть задано несколько ограничений CHECK, можно понять, что при формировании имен ограничений системой все ограничения, заданные на одной и той же таблице, приобретают примерно такие имена:

```
CK__Customers__22AA2996
CK__Customers__25869641
CK__Customers__267ABA7A
```

Вполне очевидно, что если возникнет необходимость отредактировать одно из этих ограничений, будет очень сложно выяснить, к чему относится каждое из них.

Сам автор стремится явно задавать имя для каждого ограничения и, подготавливая такое имя, задает сокращенное обозначение типа ограничения, а затем либо применяет краткую формулировку назначения этого ограничения, либо указывает имя (имена) столбца (столбцов), на который распространяется данное ограничение. Например, для ограничения, гарантирующего, чтобы пользователи приложения не продавали товары с убытком, по цене ниже стоимости, может быть задано ограничение с именем CKPriceExceedsCost, а для столбца, в котором должно быть обеспечено наличие телефонных номеров, отформатированных должным образом, может быть достаточно применить такое простое имя, как CKCustomerPhoneNo.

В настоящей книге уже говорилось о том, какими правилами следует руководствоваться при выборе имен различных объектов, но отметим еще раз, что в действительности не так важен выбор самих имен, как соблюдение перечисленных ниже требований.

- Обеспечение единообразия.
- Применение имен, понятных для всех.
- Применение наиболее краткой формулировки для имен и вместе с тем соблюдение двух указанных правил.

И еще раз подчеркнем необходимость соблюдения единообразия имен.

Ограничения ключей

В процессе эксплуатации базы данных чаще всего приходится сталкиваться с ключами четырех типов. К ним относятся первичные, внешние, альтернативные и инверсные ключи. В настоящей главе будут рассматриваться только первые три типа из указанных четырех, поскольку именно они лежат в основе ограничений, налагаемых на базу данных.

Инверсные ключи по существу представляют собой любые индексы (индексы будут рассматриваться в главе 9), которые не налагают на таблицу ограничения в той или иной форме (т.е. ограничения первичного ключа, внешнего ключа или ограничения уникальности). Инверсные ключи служат не для принудительного применения средств обеспечения целостности данных, а просто предоставляют альтернативный способ сортировки данных.

Понятие ключей относится к основополагающим понятиям проектирования и управления базой данных, поэтому относятся также к числу наиболее важных концепций, представленных в данной книге. Полное освоение концепции ключа является необходимым для успешного перехода к теме нормализации, которая представлена в главе, посвященной проектированию.

Ограничения PRIMARY KEY

Прежде чем перейти к анализу определения первичного ключа, сделаем небольшое отступление, чтобы вкратце обсудить тему, что представляют собой реляционные базы данных. В основе реляционных баз данных лежит стремление обеспечить реляционную связь между данными (т.е. связь на уровне отношений между множествами). Поэтому важным требованием к реляционным базам данных является то, что большинство таблиц (за очень редкими исключениями) должно иметь уникальный идентификатор для каждой строки. Уникальный идентификатор позволяет надежно устанавливать связь между строками различных таблиц в базе данных и тем самым формировать отношение между двумя таблицами.

В базах данных, которые эксплуатировались в среде мэйнфреймов в 1980-х и в начале 1990-х годов, а также в базах данных ISAM (dBase, FoxPro, Clipper и т.д.) применялся совершенно другой подход. В подобной среде одновременно рассматривается только одна строка. Работа с базой данных обычно заключается в том, что открывается вся таблица и последовательно просматриваются ее строки до тех пор, пока не обнаруживается искомая. Если затем оказывается, что нужно получить данные еще из одной таблицы, отдельно открывается эта таблица и осуществляется выборка данных из таблицы. А в конечном итоге для обработки зависимых друг от друга данных применяется программа.

Первичные ключи представляют собой уникальные идентификаторы для каждой строки. Столбец первичного ключа должен содержать уникальные значения (и поэтому в этом столбце не допускается наличие NULL-значения). Первичные ключи очень важны для нормальной эксплуатации реляционной базы данных, поэтому являются наиболее фундаментальными объектами базы данных по сравнению со всеми прочими ключами и ограничениями.

Не следует путать первичный ключ, который однозначно идентифицирует каждую строку в таблице, с идентификатором GUID, который является более универсальным средством, как правило, применяемым для идентификации любых объектов (а не просто строк) во времени и пространстве. Безусловно, идентификаторы GUID могут использоваться в качестве первичного ключа, но это связано с некоторыми дополнительными издержками, к тому же обычно в их применении нет особого смысла, если речь идет лишь о том, чтобы предоставить пользователю содержимое некоторой таблицы. В действительности необходимость в использовании идентификаторов GUID возникает лишь в том широко известном случае, когда создается среда базы данных с тиражируемыми или иным образом распределенными данными. В подобных случаях невозможно обойтись без применения идентификаторов GUID для создания первичного ключа.

Таблица может иметь не больше одного первичного ключа. Кроме того, как уже было сказано выше, в базе данных редко встречаются такие таблицы, для которых не требуется первичный ключ.

Следует подчеркнуть высказанную выше мысль, отметив, что таблицы без первичного ключа встречаются не просто редко, а очень редко. Наличие в базе данных таблицы, не имеющей первичного ключа, полностью противоречит представлениям о реляционном характере данных, хранимых в базе данных, поскольку из этого следует, что возможность установить связь с какой-либо конкретной строкой этой таблицы отсутствует. Данные, представленные в таблице, не имеющей первичного ключа, не имеют также каких-либо отличительных признаков.

Безусловно, нередко встречаются ситуации, в которых многочисленные строки таблицы логически идентичны друг другу, но это не означает, что возможность создания первичного ключа для такой таблицы отсутствует. В подобных обстоятельствах может быть предусмотрено искусственное создание ключа того или иного типа. Подобный подход чаще всего реализуется с использованием столбца идентификации, но в некоторых ситуациях в большей степени оправдано применение идентификаторов GUID.

Первичный ключ гарантирует уникальность сочетания значений столбцов, объявленных как принадлежащие к этому первичному ключу. Сами эти уникальные значения служат в качестве идентификаторов для каждой строки в таблице. Для создания первичного ключа по существу применяются два способа. Первичный ключ может быть либо создан с помощью команды `CREATE TABLE` во время создания таблицы, либо введен в действие впоследствии с помощью команды `ALTER TABLE`.

Создание первичного ключа при создании таблицы

Рассмотрим один из операторов `CREATE TABLE`, приведенных в предыдущей главе:

```
CREATE TABLE Customers
(
   CustomerNo      int          IDENTITY    NOT NULL,
   CustomerName    varchar(30)              NOT NULL,
   Address1        varchar(30)              NOT NULL,
   Address2        varchar(30)              NOT NULL,
   City            varchar(20)              NOT NULL,
   State           char(2)                  NOT NULL,
   Zip             varchar(10)              NOT NULL,
   Contact         varchar(25)              NOT NULL,
   Phone           char(15)                 NOT NULL,
   FedIDNo         varchar(9)               NOT NULL,
   DateInSystem    smalldatetime            NOT NULL
)
```

Читатель уже должен быть хорошо знаком с подобными операторами `CREATE`, а теперь введем в него очень важную часть — ограничение `PRIMARY KEY`. В качестве первичного ключа обозначим столбец `CustomerNo`. Подробное описание критериев выбора наиболее подходящих первичных ключей приведено в следующей главе, а на данный момент отметим, что решение по выбору столбца `CustomerNo` может быть обосновано хотя бы тем, что необходимо обеспечить уникальность значений идентификаторов заказчиков в столбце `CustomerNo`. Подобные системы, в которых для каждого из обслуживаемых заказчиков предусмотрен уникальный идентификатор, эксплуатируются уже много лет, поэтому нет смысла снова изобретать это колесо.

Чтобы внести корректировку в рассматриваемый оператор CREATE TABLE для задания в нем ограничения PRIMARY KEY, достаточно ввести информацию об ограничении сразу после определения столбца (столбцов), который должен войти в состав первичного ключа. В данном случае достаточно задать ключевое слово PRIMARY KEY:

```
CREATE TABLE Customers
(
  CustomerNo     int          IDENTITY    NOT NULL
    PRIMARY KEY,
  CustomerName   varchar(30)              NOT NULL,
  Address1       varchar(30)              NOT NULL,
  Address2       varchar(30)              NOT NULL,
  City           varchar(20)              NOT NULL,
  State          char(2)                  NOT NULL,
  Zip            varchar(10)              NOT NULL,
  Contact        varchar(25)              NOT NULL,
  Phone          char(15)                 NOT NULL,
  FedIDNo        varchar(9)               NOT NULL,
  DateInSystem   smalldatetime            NOT NULL
)
```

Следует отметить, что для проверки этого кода необходимо вначале удалить существующую таблицу с помощью команды DROP TABLE Customers. Кроме того, внесенные в оператор изменения сводятся к тому, что модифицирована одна строка (удалена запятая в конце) и добавлен небольшой объем кода во вторую строку, относящуюся к этому столбцу. Иными словами, доработка оказалась несложной! Еще раз отметим, что все изменение свелось к добавлению одного простого ключевого слова (разумеется, оно складывается из двух слов, но они рассматриваются как одно ключевое слово), после чего таблица стала обладать первичным ключом.

Создание первичного ключа на существующей таблице

Кроме объявления таблицы с первичным ключом, предусмотрена возможность создания первичного ключа на существующей таблице. Это — также несложная задача. Рассмотрим, как она решается, на примере таблицы Employees:

```
USE Accounting
ALTER TABLE Employees
  ADD CONSTRAINT PK_EmployeeID
  PRIMARY KEY (EmployeeID)
```

Ниже описано, какие сведения передаются в СУБД SQL в рассматриваемой команде ALTER.

- Указание на то, что должно быть сделано дополнение к определению таблицы (при желании можно было бы также исключить какую-то часть существующего определения таблицы).
- Уточнение дополняемой части определения (согласно которому вводится ограничение).
- Имя, предусмотренное для ограничения (которое позволит в дальнейшем непосредственно обращаться к ограничению).
- Тип ограничения (PRIMARY KEY).
- Столбец (столбцы), на который распространяется ограничение.

Ограничения FOREIGN KEY

Внешние ключи не только обеспечивают целостность данных, но и создают связи между таблицами. После задания внешнего ключа на таблице устанавливается зависимость между таблицей, для которой определяется внешний ключ (так называемой **ссылающейся** таблицей), и таблицей, на которую ссылается внешний ключ (так называемой таблицей, **упомянутой в ссылке**). После задания на ссылающейся таблице внешнего ключа любая строка, вставляемая в эту таблицу, должна соответствовать одному из следующих двух условий: иметь согласующуюся с ней строку в столбце (столбцах), которому соответствует внешний ключ таблицы, указанной в ссылке, или иметь значение столбца (столбцов) внешнего ключа, равное NULL. Такое определение может показаться немного запутанным, поэтому рассмотрим, как оно применяется на примере.

В предыдущем предложении слова "иметь значение, равное NULL", указывают на то, что фактически должен содержать оператор INSERT, применяемый к ссылающейся таблице. Как будет вскоре описано, в действительности эти данные после ввода в таблицу могут выглядеть немного иначе, в зависимости от того, какие опции заданы в объявлении FOREIGN KEY.

Создадим в базе данных Accounting еще одну таблицу и назовем ее Orders. В приведенном ниже сценарии с оператором CREATE заслуживает внимания то, что в таблицах предусмотрено применение и первичного, и внешнего ключа. Из дальнейшего описания рассматриваемого проекта станет ясно, что существенно важной частью объявления одной из таблиц является спецификация первичного ключа. С другой стороны, объявление внешнего ключа, который задается на таблице в этом сценарии, почти полностью совпадает с объявлением первичного ключа, не считая того, что в этом объявлении указана таблица, на которую ссылается внешний ключ. Синтаксическая структура объявления внешнего ключа предусматривает необходимость указания столбца или столбцов, на которые распространяется ограничение FOREIGN KEY, и выглядит примерно таким образом:

```
<column name> <data type> <nullability>
FOREIGN KEY REFERENCES <table name>(<column name>)
    [ON DELETE {CASCADE|NO ACTION|SET NULL|SET DEFAULT}]
    [ON UPDATE {CASCADE|NO ACTION|SET NULL|SET DEFAULT}]
```

Практическое занятие

Создание таблицы с внешним ключом

При выполнении этого задания не предусмотрено использование конструкции ON. Таким образом, применительно к таблице Orders может быть подготовлен сценарий, который выглядит, как показано ниже.

```
USE Accounting
CREATE TABLE Orders
(
  OrderID     int   IDENTITY    NOT NULL
    PRIMARY KEY,
  CustomerNo  int               NOT NULL
    FOREIGN KEY REFERENCES Customers(CustomerNo),
  OrderDate   smalldatetime     NOT NULL,
  EmployeeID  int                NOT NULL
)
```

Следует учитывать, что столбец, на который фактически ссылается внешний ключ, должен иметь определенное на нем ограничение PRIMARY KEY или UNIQUE (ограничения UNIQUE описаны ниже в данной главе).

Следует также отметить, что на одном и том же столбце могут быть заданы одновременно и первичный, и внешний ключ. Пример такого применения ключей можно видеть в таблице `Order Details` *базы данных* `Northwind`. *В этой таблице первичный ключ состоит из столбцов* `OrderID` *и* `ProductID`, *а на обоих этих столбцах заданы также внешние ключи, которые ссылаются соответственно на таблицы* `Orders` *и* `Products`. *В настоящей главе также будет фактически создана таблица, имеющая столбец, на котором заданы одновременно и первичный, и внешний ключ.*

Описание полученных результатов

После успешного применения приведенного выше кода вызовите на выполнение процедуру sp_help. После этого должно быть обнаружено, что в разделе constraints тех результатов, которые получены с помощью процедуры sp_help, будет содержаться информация о новом ограничении. Кроме того, для получения еще более подробных сведений об имеющихся ограничениях можно вызвать на выполнение процедуру sp_helpconstraint, которая также имеет простой синтаксис:

```
EXEC sp_helpconstraint <table name>
```

После применения процедуры sp_helpconstraint к вновь созданной таблице Orders будет получена информация, позволяющая узнать имена, условия и состояние всех ограничений, заданных на этой таблице. Очевидно, что на данном этапе таблица Orders имеет одно ограничение FOREIGN KEY и одно ограничение PRIMARY KEY.

> В результатах применения процедуры sp_helpconstraint к рассматриваемой таблице можно обнаружить слово (clustered) непосредственно после сведений об ограничении PRIMARY KEY; оно просто означает, что данный индекс является кластеризованным. Смысл этого понятия будет раскрыт более подробно в главе 9.

Итак, определение внешнего ключа было задано непосредственно в объявлении таблицы. Таким образом, после создания этой таблицы внешний ключ становится ее неотъемлемой частью. Как было указано в главе 1, поддержание собственной целостности должна обеспечивать сама база данных; каждый внешний ключ принудительно вводит одно из ограничений, распространяющихся на хранимые данные, и поэтому гарантирует обеспечение целостности базы данных.

В отличие от первичных ключей, количество внешних ключей, заданных на таблице, не должно ограничиваться только одним. Для любой таблицы может быть задано от нуля до 253 внешних ключей. Единственным условием является то, что каждый конкретный столбец может упоминаться только в одном внешнем ключе. Тем не менее в каждом отдельном внешнем ключе может быть задано несколько столбцов. Кроме того, предусмотрена также возможность использовать какой-то конкретный столбец в качестве назначения ссылок, заданных во внешних ключах многих таблиц.

Добавление внешнего ключа к существующей таблице

На практике иногда возникают ситуации, когда требуется дополнительно задать на таблице не только первичный ключ, но и какие-либо другие ограничения, в том числе внешние ключи. Для этого применяется процедура, аналогичная созданию первичного ключа.

Практическое занятие

Добавление внешнего ключа к существующей таблице

Зададим еще один внешний ключ на таблице `Orders`, чтобы обеспечить ввод в поле `EmployeeID` (предназначенное для хранения идентификаторов служащих, которые вводят заказы) только действительных данных о служащих, которые определены в таблице `Employees`. Для этого необходимо предусмотреть уникальную идентификацию целевой строки в таблице, указанной в ссылке. Как уже было отмечено выше, этой цели можно достичь, ссылаясь либо на первичный ключ, либо на столбец с ограничением `UNIQUE`. В данном случае воспользуемся существующим первичным ключом, который был задан на таблице `Employees` в одном из примеров, приведенных выше в настоящей главе, следующим образом:

```
ALTER TABLE Orders
   ADD CONSTRAINT FK_EmployeeCreatesOrder
   FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
```

После этого еще раз вызовите на выполнение процедуру `sp_helpconstraint` применительно к таблице `Orders` и убедитесь в том, что новое ограничение было успешно задано.

Описание полученных результатов

Последнее введенное в этом примере ограничение действует точно так же, как и предыдущее, — регламентирует правила, применяемые к данным, вводимым в соответствующий столбец, согласно объявлению самой таблицы. Определение столбца не позволяет, например, вводить строковые данные в столбец числового типа, а ограничение не дает возможности вставлять в таблицу `Orders` такую строку, в которой значение поля `Employee` с данными о служащем, ответственным за ввод заказа, не ссылается на действительное значение `EmployeeID`. Если кто-то попытается ввести строку, которая не согласуется со строкой данных о служащем, такая операция вставки в таблицу `Orders` будет отвергнута в целях обеспечения целостности базы данных.

Обратите внимание на то, что даже после задания двух внешних ключей в последней части результатов выполнения процедуры `sp_helpconstraint` (или вкладки Messages, *если выбрана опция* Results in Grid*) по-прежнему появляется строка со словами "`No foreign keys reference this table`", которые свидетельствуют о том, что в базе данных отсутствуют какие-либо другие таблицы, ссылающиеся на данную таблицу, хотя в самой этой таблице имеются внешние ключи, ссылающиеся на другие таблицы. Чтобы убедиться в том, что в базе данных учитывается различие между ссылающимися таблицами и таблицами, указанными в ссылке, вызовите на данном этапе процедуру `sp_helpconstraint` применительно к таблице `Customers` или `Employees` и убедитесь в том, что для каждой из этих таблиц указано, что на нее ссылается новая таблица `Orders`.*

Создание таблицы, ссылающейся на саму себя

Иногда возникает необходимость задать в ограничении столбец, находящийся не в другой таблице, а непосредственно в той же таблице, в которой создается ссылка на исходное ограничение. Это означает, что одна и та же таблица применительно к некоторому ограничению может играть роль и ссылающейся таблицы, и таблицы, указанной в ссылке. Разумеется, подобные ситуации встречаются не очень часто, но достаточно регулярно.

Прежде чем фактически создать ограничение, ссылающееся на столбец из той же таблицы, в котором упоминается обязательный (не допускающий применения неопределенных значений) столбец, допустим, представляющий собой столбец идентификации, крайне необходимо ввести в таблицу по меньшей мере одну строку и только после этого задавать внешний ключ. Такое требование обусловлено тем, что идентификационное значение выбирается и заполняется после того, как уже была осуществлена проверка внешнего ключа и принудительно введено заданное в нем ограничение. Это означает, что при отсутствии строк в таблице еще отсутствует значение, на которое могла бы указывать ссылка из первой строки при проведении проверки. Из этой ситуации есть также еще один выход – начать с создания требуемого внешнего ключа, а затем запретить его использование при вводе первой строки. Дополнительные сведения о том, как можно отменить использование ограничений, приведены ниже в этой главе.

Итак, в данном случае рассматривается таблица, в которой имеется ссылка на столбец, представляющий собой столбец идентификации, поэтому необходимо вначале ввести в таблицу хотя бы одну первичную строку и только после этого задавать ограничение:

```
INSERT INTO Employees
   (
   FirstName,
   LastName,
   Title,
   SSN,
   Salary,
   PriorSalary,
   HireDate,
   ManagerEmpID,
   Department
   )
VALUES
   (
   'Billy Bob',
   'Boson',
   'Head Cook & Bottle Washer',
   '123-45-6789',
   100000,
   80000,
   '1990-01-01',
   1,
   'Cooking and Bottling'
   )
```

Теперь, после ввода первичной строки, можно приступить к заданию внешнего ключа. В этом варианте создания таблицы, ссылающейся на саму себя, в котором ис-

пользуется оператор ALTER, осуществляемые действия аналогичны тем, которые выполняются при уточнении любого другого определения внешнего ключа. Проверим действие следующего оператора на практике:

```
ALTER TABLE Employees
   ADD CONSTRAINT FK_EmployeeHasManager
   FOREIGN KEY (ManagerEmpID) REFERENCES Employees(EmployeeID)
```

В данном операторе есть только одно отличие от оператора CREATE. Но есть и еще один нюанс, состоящий в том, что в данном определении допускается не использовать ключевое слово FOREIGN KEY (но этого не следует делать) и оставлять только конструкцию REFERENCES. К этому времени таблица Employees уже определена, но если бы речь шла о ее создании с самого начала, то на данном этапе можно было бы применить следующий сценарий (особого внимания заслуживает определение внешнего ключа на столбце ManagerEmpID):

```
CREATE TABLE Employees (
   EmployeeID          int          IDENTITY    NOT NULL
      PRIMARY KEY,
   FirstName           varchar (25)             NOT NULL,
   MiddleInitial       char (1)                 NULL,
   LastName            varchar (25)             NOT NULL,
   Title               varchar (25)             NOT NULL,
   SSN                 varchar (11)             NOT NULL,
   Salary              money                    NOT NULL,
   PriorSalary         money                    NOT NULL,
   LastRaise AS Salary - PriorSalary,
   HireDate            smalldatetime            NOT NULL,
   TerminationDate     smalldatetime            NULL,
   ManagerEmpID        int                      NOT NULL
      REFERENCES Employees(EmployeeID),
   Department          varchar (25)             NOT NULL
)
```

Следует отметить, что при попытке уничтожить таблицу Employees в данный момент (чтобы выполнить оператор, рассматриваемый во втором примере) было бы получено сообщение об ошибке. Это связано с тем, что после определения в таблице Orders ссылки на таблицу Employees эти две таблицы становятся, как принято выражаться, "связанными со схемой". Это означает, что в базе данных теперь содержится информация о наличии так называемой зависимости другой таблицы от таблицы Employees. С учетом такой информации СУБД SQL Server не позволяет уничтожать таблицу, на которую ссылается другая таблица. Чтобы получить возможность выполнить с помощью СУБД SQL Server удаление таблицы Employees (или, по той же причине, таблицы Customers), необходимо вначале уничтожить внешний ключ в таблице Orders.

Кроме того, необходимо учитывать, что таблица, созданная с объявлением внешнего ключа, ссылающимся на ту же таблицу в ограничении, не позволит ввести даже первоначальную строку, поэтому подобные объявления следует применять лишь с учетом того условия, чтобы ограничение внешнего ключа было задано на столбце, допускающем ввод NULL-значений. Таким образом появится возможность ввести первую строку, имеющую NULL-значение, в столбце внешнего ключа и тем самым избежать необходимости задания исходной строки.

Каскадное осуществление действий

Одним из важных различий между внешними ключами и ключами других типов является то, что внешние ключи двунаправлены. Иными словами, действие внешних ключей выражается не только в том, что они обусловливают ввод в дочернюю таблицу только таких значений, которые представлены в родительской таблице, но и обеспечивают проверку строк дочерней таблицы при внесении изменений в родительскую таблицу (что позволяет предотвратить появление в дочерней таблице так называемых *висячих строк*, т.е. строк, потерявших связь со строками родительской таблицы). По умолчанию СУБД SQL Server "защищает" от удаления такие строки родительской таблицы, которым соответствуют строки, существующие в дочерней таблице. Но иногда предпочтительный способ организации работы приложения состоит в том, чтобы удаление всех зависимых строк, препятствующих удалению строки, на которую они ссылаются, происходило автоматически. Аналогичный принцип применяется к обновлению строк, поскольку требуется, чтобы в зависимых строках автоматически восстанавливалась ссылка на вновь обновленные строки. Немного реже встречается ситуация, в которой требуется перевести ссылающуюся строку в какое-то другое заведомо известное состояние. В последнем случае имеется возможность либо задать в зависимой строке NULL-значение, либо применить то значение, которое предусмотрено по умолчанию, для соответствующего столбца.

Процесс автоматического осуществления подобных операций удаления и обновления известен под названием **каскадного выполнения действий**. Фактически этот процесс, особенно применительно к операциям удаления, может проходить через несколько уровней зависимостей (когда одна строка зависит от другой, вторая зависит от третьей и т.д.). Каскадное выполнение действий предусмотрено и в СУБД SQL Server. Для использования этой возможности достаточно прибегнуть к дополнительным опциям синтаксических конструкций при объявлении внешнего ключа — ввести те конструкции `ON`, которые были пропущены в начале этого раздела.

Рассмотрим возможность применения каскадных действий на примере еще одной таблицы, создаваемой в базе данных `Accounting`. Назовем эту таблицу `OrderDetails` и будем использовать ее для хранения отдельных элементов с обозначением товаров в заказе:

```
CREATE TABLE OrderDetails
(
   OrderID         int               NOT NULL,
   PartNo          varchar(10)       NOT NULL,
   Description     varchar(25)       NOT NULL,
   UnitPrice       money             NOT NULL,
   Qty             int               NOT NULL,
   CONSTRAINT      PKOrderDetails
      PRIMARY KEY  (OrderID, PartNo),
   CONSTRAINT      FKOrderContainsDetails
      FOREIGN KEY  (OrderID)
         REFERENCES Orders(OrderID)
         ON UPDATE NO ACTION
         ON DELETE CASCADE
)
```

В приведенном выше операторе используется много новых конструкций, поэтому рассмотрим отдельно все не встречавшиеся ранее компоненты.

Прежде чем приступить к подробному описанию объявления внешнего ключа в этом операторе, остановимся на том, как объявлен в нем первичный ключ. Автор решил не задавать объявление ограничения непосредственно после объявления самого ключа, а выделить его в отдельную конструкцию `constraint`. Такая структура объявления, во-первых, упрощает работу при создании первичного ключа, состоящего из нескольких столбцов (ведь первичный ключ с несколькими столбцами не может быть объявлен как ограничение столбца), и, во-вторых, позволяет проще понять общее устройство оператора `CREATE TABLE`. Аналогичным образом, можно было бы задать объявление внешнего ключа непосредственно вслед за объявлением столбца. Но автор предпочел другой возможный вариант и также применил отдельную конструкцию `constraint`. Дополнительная информация по этой теме приведена ниже в данной главе.

Прежде всего необходимо учесть, что применяемый внешний ключ является также частью первичного ключа. Такая организация дочерних таблиц встречается далеко не так уж редко, а в случае ассоциативных таблиц (которые рассматриваются более подробно в следующей главе) применяется почти всегда. Но следует помнить, что каждое ограничение всегда действует автономно, поэтому добавление. модификация или удаление каждого ограничения осуществляется независимо от других ограничений.

Теперь рассмотрим само объявление внешнего ключа:

```
FOREIGN KEY  (OrderID)
REFERENCES   Orders(OrderID)
```

Столбец `OrderID` был объявлен как зависящий от "внешнего" столбца. В данном случае таковым является столбец (имеющий совпадающее с ним имя, `OrderID`) отдельной таблицы (`Orders`), но, как было показано выше в данной главе, в случае необходимости можно было бы с таким же успехом выбрать какой-то столбец из той же таблицы.

При создании внешних ключей, ссылающихся на ту же таблицу, в которой определен внешний ключ, необходимо учитывать некоторые нюансы. В частности, в объявлении внешних ключей такого типа не допускается использование декларативных действий `CASCADE`. Такое условие предусмотрено для того, чтобы можно было избежать возникновения циклических операций обновления или удаления, т.е. таких ситуаций, в которых обновление одной строки вызывает обновление другой, а это, в свою очередь, приводит к необходимости обновить первую из этих строк. В результате может возникнуть бесконечный цикл.

Теперь приступим к изучению конструкций `ON`, чтобы лучше разобраться в проблеме каскадного осуществления действий:

```
ON UPDATE   NO ACTION
ON DELETE   CASCADE
```

В этих двух конструкциях определены два различных **действия по обеспечению ссылочной целостности**. Вполне очевидно, что действием по обеспечению ссылочной целостности является такое действие, которое должно осуществляться при каждом вызове правила обеспечения ссылочной целостности. Для тех ситуаций, когда происходит обновление строки родительской таблицы (таблицы `Orders`), указано,

что такие обновления не должны распространяться каскадно на дочернюю таблицу (OrderDetails). Но в иллюстративных целях автор выбрал для операций удаления действие CASCADE.

Следует отметить, что опция NO ACTION применяется по умолчанию, поэтому мы не были обязаны задавать ее в приведенном выше коде. Но данное ключевое слово не поддерживалось в версиях, предшествующих SQL Server 2000, и это стало причиной такого положения дел, что стало "общепринятым" не задавать в коде ключевое слово NO ACTION. Но если поддержка обратной совместимости не требуется, то автор рекомендует явно задавать ключевое слово NO ACTION, чтобы явно выразить в коде свои намерения.

Предпримем попытку выполнить вставку в таблицу OrderDetails следующим образом:

```
INSERT INTO OrderDetails
VALUES
   (1, '4X4525', 'This is a part', 25.00, 2)
```

Но если до сих пор не была выполнена вставка каких-либо данных в эту таблицу, осуществление такой попытки приведет к возникновению следующей ошибки:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the FOREIGN KEY constraint
"FKOrderContainsDetails". The conflict occurred in database "Accounting",
table "Orders", column 'OrderID'.
The statement has been terminated.
```

Это связано с тем, что в таблице Orders отсутствуют какие-либо данные, поэтому невозможно сформировать ссылку на какую-либо строку в таблице Orders, поскольку в этой таблице вообще нет никаких строк.

В данном примере впервые в настоящей книге иллюстрируется одна из сложностей, возникающих в процессе эксплуатации реляционной базы данных, – цепочки зависимостей. Цепочка зависимостей обнаруживается в таких ситуациях, когда выполнение операции над некоторым объектом зависит от результатов операции над каким-то другим объектом, что может зависеть от результатов работы еще над каким-то объектом, и т.д. При этом фактически пользователь не может найти приемлемый выход из положения, поскольку возникновение указанной ситуации обусловлено структурой самой базы данных. Поэтому пользователь обязан перейти к началу цепочки и выполнять все необходимые в ней действия до тех пор, пока не сможет осуществить действительно требуемую ему операцию. Но, к счастью, чаще всего результаты, которые должны быть получены на промежуточных этапах прохождения цепочки зависимостей, уже в основном имеются в базе данных, пусть даже на одном или двух уровнях зависимостей.

Итак, в данном случае вставка строки в таблицу OrderDetails может быть осуществлена только при том условии, что в таблице Orders уже будет хотя бы одна строка. Но, к сожалению, для ввода строки в таблицу Orders требуется, чтобы в таблице Customers также была хотя бы одна строка (напомним, что на таблице Orders сформирован внешний ключ). Поэтому вставка требуемой строки должна проводиться поэтапно, начиная, допустим, с выполнения следующего оператора:

```
INSERT INTO Customers   -- Данные о заказчике; следует учитывать, что
                        -- столбец CustomerNo - столбец идентификации
VALUES
   ('Billy Bob''s Shoes',
```

```
'123 Main St.',
' ',
'Vancouver',
'WA',
'98685',
'Billy Bob',
'(360) 555-1234',
'931234567',
GETDATE()
)
```

Вставка данных о заказчике должна быть выполнена успешно, но, чтобы убедиться в этом, выполним выборку соответствующей строки, как показано в табл. 6.2.

Таблица 6.2. Результаты выборки одной записи из таблицы `Customers`

Поле	Значение
Customer No	1
Customer Name	Billy Bob's Shoes
Address 1	123 Main Street
Address 2	
City	Vancouver
State	WA
Zip	98685
Contact	Billy Bob
Phone	(360) 555-1234
FedIDNo	931234567
DateInSystem	2000-07-1021:17:00

Таким образом, в таблице `Customers` уже имеются сведения о заказчике с идентификатором `CustomerID`, равным 1 (если с таблицей `Customers` проводились какие-либо эксперименты, то в действительности идентификатор заказчика может оказаться другим). Теперь появилась возможность взять числовое значение идентификатора и применить его в следующем операторе `INSERT` (который наконец-то будет применимым к таблице `Orders`). Итак, выполним вставку данных о заказе, относящихся к заказчику с идентификатором `CustomerID`, равным 1:

```
INSERT INTO Orders
   (CustomerNo, OrderDate, EmployeeID)
VALUES
   (1, GETDATE(), 1)
```

На этот раз выполнение операции вставки должно завершиться успешно.

> Следует отметить, что в таблицу `Employees` уже была вставлена первоначальная строка, поэтому проведение указанной выше операции вставки не вызвало появления еще одной ошибки; в противном случае пришлось бы вначале вставить строку в таблицу `Employees` и только после этого СУБД SQL Server позволила бы ввести необходимую строку в таблицу `Orders` (напомним, что на таблице `Employees` задан внешний ключ).

И только после этого появляется возможность осуществить вставку данных в таблицу OrderDetails. Но для того чтобы пример, в котором рассматривается конструкция CASCADE, стал более наглядным, будут вставлены две строки, а не одна:

```
INSERT INTO OrderDetails
VALUES
  (1, '4X4525', 'This is a part', 25.00, 2)
INSERT INTO OrderDetails
VALUES
  (1, '0R2400', 'This is another part', 50.00, 2)
```

Теперь приступим к проверке результатов, выполнив следующий оператор SELECT:

```
SELECT OrderID, PartNo FROM OrderDetails
```

Это, как и следовало ожидать, приведет к получению таких двух строк:

```
OrderID     PartNo
---------   ---------
1           0R2400
1           4X4525

(2 row(s) affected)
```

На данном этапе подготовлены все данные, необходимые для проверки действия конструкции CASCADE, поэтому приступим к экспериментам. Для этого удалим одну строку из таблицы Orders и посмотрим, какое действие окажет эта операция удаления на таблицу OrderDetails:

```
USE Accounting
-- Вначале рассмотрим строки в обеих таблицах
SELECT *
FROM Orders
SELECT *
FROM OrderDetails
-- Затем удалим строку Order
DELETE Orders
WHERE OrderID = 1
-- Наконец, еще раз рассмотрим оба набора данных и определим, в чем
-- состоит результат каскадного действия
SELECT *
FROM Orders
SELECT *
FROM OrderDetails
```

В конечном итоге вырабатываются такие интересные результаты:

```
OrderID     CustomerNo      OrderDate              EmployeeID
---------   ------------    --------------------   --------------
1           1               2000-07-13 22:18:00    1

(1 row(s) affected)
OrderID     PartNo     Description            UnitPrice     Qty
---------   -------    --------------------   ------------  ----
1           0R2400     This is another part   50.0000       2
1           4X4525     This is a part         25.0000       2

(2 row(s) affected)
```

```
(1 row(s) affected)
OrderID     CustomerNo    OrderDate          EmployeeID
--------    -----------   ------------       ----------------------

(0 row(s) affected)
OrderID     PartNo        Description        UnitPrice     Qty
----------  -----------   ---------------    ------------  --------

(0 row(s) affected)
```

Обратите внимание на то, что в операторе DELETE была указана только таблица Orders, но несмотря на это, действие операции удаления распространилось также каскадным путем на строки таблицы OrderDetails, согласующиеся с удаляемой строкой, поэтому произошло удаление строк из обеих таблиц. Если бы таблица OrderDetails была определена с конструкцией CASCADE, касающейся обновления, и произошло обновление соответствующей строки, то результаты операции обновления также были бы распространены на эту дочернюю таблицу, OrderDetails.

Следует учитывать, что пределы глубины, на которую могут распространять действия конструкции CASCADE, не установлены. Например, если бы была объявлена таблица ShipmentDetails с конструкцией CASCADE, которая ссылается на строки таблицы OrderDetails, то произошло бы удаление строк и из таблицы ShipmentDetails в результате применения лишь одного оператора DELETE к таблице Orders.

В действительности этот аспект использования каскадных действий является одним из самых опасных. Дело в том, что иногда бывает очень трудно понять, к каким последствиям для базы данных может привести выполнение единственного оператора DELETE или UPDATE. По этой и по другим причинам автор не вполне одобряет применение каскадных действий. Рассчитывая на них, программисты пытаются сэкономить свои усилия по разработке кода, но каскадное осуществление действий не всегда приводит к наилучшим последствиям, особенно когда речь идет об удалении данных!

Другие варианты действий, осуществляемых с помощью конструкции CASCADE

Выше в данной главе были приведены примеры каскадных обновлений и удалений, но в определении конструкции CASCADE предусмотрены еще два вида каскадных действий — SET NULL и SET DEFAULT.

Эти ключевые слова были впервые введены в версии SQL Server 2005, поэтому, если необходимо обеспечить обратную совместимость с программным обеспечением SQL Server 2000, следует избегать их применения. Однако в целом соответствующие действия осуществляются очень просто: если выполняется операция обновления, в результате которой изменяется значение в строке родительской таблицы по отношению к другой строке, то в строке дочерней таблицы задается либо NULL-значение, либо значение, предусмотренное по умолчанию для соответствующего столбца (в зависимости от того, выбрано ли ключевое слово SET NULL или SET DEFAULT). Результаты применения указанных ключевых слов сводятся только к этому.

Другие вопросы, связанные с использованием внешних ключей

Прежде чем завершить описание внешних ключей, необходимо затронуть некоторые другие темы. Безусловно, в настоящей книге мы будем снова и снова возвращаться к проблематике применения внешних ключей, но на данный момент необходимо отметить ряд дополнительных нюансов:

- под влиянием чего значения в столбцах внешних ключей могут становиться обязательными или необязательными;
- благодаря чему действие внешних ключей становится двунаправленным.

Причины, по которым значения в столбцах внешних ключей могут становиться обязательными или необязательными

Согласно самому определению внешнего ключа, предусмотрены два указанных ниже возможных варианта заполнения данными столбца (или столбцов), на котором определен внешний ключ.

- Заполнение столбца значениями, которое согласуются со значениями соответствующего столбца в таблице, указанной в ссылке.
- Полный отказ от заполнения столбца какими-либо действительными значениями и ввод вместо них NULL-значений.

Прежде всего можно предусмотреть такой вариант заполнения столбца внешнего ключа, чтобы он стал полностью обязательным (в результате чего пользователи таблицы с внешним ключом будут вынуждены применять только первый вариант из приведенного выше списка). Для этого достаточно определить ссылающийся столбец как имеющий конструкцию `NOT NULL`. Таким образом, в столбце родительской таблицы не будет разрешено использование NULL-значений, а поскольку столбец внешнего ключа будет обязан содержать только значения, отличные от `NULL`, чтобы обеспечить согласование со значениями в таблице, указанной в ссылке, то заведомо удастся обеспечить согласование каждой строки ссылающейся таблицы хотя бы с одной из строк таблицы, указанной в ссылке. Иными словами, связь между родительской и дочерней таблицами становится обязательной.

Если же допускается, чтобы ссылающийся столбец содержал NULL-значения, то к таблицам предъявляются аналогичные требования, не считая того, что пользователь получает также возможность не задавать значение, поэтому допускается вставка в ссылающуюся таблицу строк, содержащих NULL-значения в столбце внешнего ключа, даже несмотря на то, что `NULL` не может быть согласован со NULL-значением в таблице, указанной в ссылке.

Причины, по которым действие внешних ключей становится двунаправленным

Мы вкратце коснулись этой темы при описании каскадных действий, но, возможно, еще недостаточно подчеркнули ту мысль, что в результате определения внешних ключей ограничения по существу налагаются на обе таблицы. Вплоть до этого момента проблематика внешних ключей рассматривалась с точки зрения использования ссылающейся таблицы, но после определения внешнего ключа приходится также соблюдать некоторые требования и при использовании таблицы, указанной в ссылке.

> По умолчанию нельзя удалять строку в таблице, указанной в ссылке, или обновлять столбец в таблице, указанной в ссылке, если на соответствующую строку указывает ссылка из зависимой таблицы. Если же есть необходимость обеспечить удаление или обновление такой строки, то для получения подобной возможности требуется предусмотреть применение каскадных действий в операциях удаления и (или) обновления.

Проанализируем более подробно принцип, согласно которому нельзя удалять или обновлять строку, указанную в ссылке.

Выше в данной главе был приведен пример, в котором было определено несколько внешних ключей для таблицы Orders. Один из этих внешних ключей ссылается на столбец EmployeeID таблицы Employees. Предположим, например, что в компании около двух лет работал служащий с идентификатором EmployeeID, равным 10, который ввел много заказов, а затем решил уволиться и перейти на другую работу. Было бы вполне резонным такое решение – удалить из таблицы Employees строку с информацией об этом служащем, но реализация подобного решения привела бы к возникновению весьма существенной проблемы, поскольку в таблице Orders появились бы так называемые *висячие строки*. Иными словами, в таблице Orders осталось бы большое количество строк, в которых по-прежнему было бы указано значение EmployeeID, равное 10. Таким образом, если бы была возможность удалить строку из таблицы Employees с идентификатором EmployeeID, равным 10, то была бы исключена другая возможность – мы больше не могли бы определить, кто из служащих ввел те или другие заказы. Это означает, что соответствующее значение в столбце EmployeeID таблицы Orders потеряло бы смысл!

Теперь проанализируем рассматриваемый пример немного более подробно. На этот раз предположим, что служащий с идентификатором 10 не уволился, а по какой-то причине, не имеющей значения в данном контексте, потребовалось сменить идентификационный номер данного служащего. Если бы такое изменение было внесено (с помощью оператора UPDATE) в таблицу Employees, а соответствующее обновление в таблице Orders не было предусмотрено, то снова появились бы висячие строки со значением 10 в столбце EmployeeID таблицы Orders, которому не соответствовал бы ни один служащий.

Продолжение анализа данного примера в том же направлении приводит к получению еще более интересных результатов! Предположим, что после этого по недосмотру произошла бы вставка новой строки со значением EmployeeID, равным 10. Это привело бы к тому, что значительное количество строк в таблице Orders оказалось бы связанным со строкой данных о служащем, который фактически не вводил эти заказы. Таким образом, в базе данных появились бы неверные данные (к сожалению).

По умолчанию СУБД SQL Server противодействует появлению висячих строк, поэтому не позволяет применять операции удаления или обновления строк к таблице, указанной в ссылке (в данном случае к таблице Employees). Такое условие перестает действовать только после удаления или обновления всех зависимых строк из ссылающейся таблицы (в данном случае таблицы Orders).

В действительности ситуации, описанные выше, наглядно иллюстрируют необходимость дополнительного обсуждения того, в каких условиях применение каскадных действий оправдано и не оправдано. Рассмотрим кратко эту тему. Поскольку для нас очень важно обеспечить целостность данных, то, по-видимому, не следует допускать удаление данных о служащем из таблицы Employees, если есть зависимые от них строки в таблице Orders, так как данные о введенных заказах отчасти теряют смысл, если невозможно определить, какой служащий ввел эти заказы. С другой стороны, операция, предусматривающая смену идентификатора служащего, может оказаться вполне допустимой (даже если может создаться впечатление, что причина для выполнения такого действия является весьма надуманной). Дело в том, что отрицательных последствий смены идентификатора служащего можно избежать, предусмотрев в таблице Orders конструкцию CASCADE, допускающую обновление. На основании изложенного можно сделать еще один вывод — применение конструкции CASCADE по отношению к операциям обновления и удаления не должно быть основано на одинаковом подходе. Возможность использования указанной конструкции в обоих этих случаях должно рассматриваться отдельно (и анализироваться очень тщательно).

Вполне очевидно, что даже после определения внешнего ключа только на одной таблице фактически налагаются конкретные условия на некоторые действия, осуществляемые применительно к обеим таблицам (а если внешний ключ ссылается на ту же таблицу, то оба набора условий распространяются на одну и ту же таблицу).

Ограничения UNIQUE

Тема, касающаяся ограничений UNIQUE, является относительно простой. По существу эти ограничения почти полностью соответствуют ограничениям первичного ключа, поскольку требуют наличия уникальных значений во всем указанном в них столбце (или в комбинации столбцов) таблицы. Ограничения UNIQUE часто называют ограничениями **альтернативных ключей**. Основные различия между ограничениями UNIQUE и ограничениями первичного ключа состоят в том, что в качестве ограничений UNIQUE обычно не применяются уникальные идентификаторы строк таблицы (даже несмотря на то, что данные в столбцах, на которых определено ограничение UNIQUE, фактически могут использоваться таким образом), а, кроме того, на таблице может быть задано несколько ограничений UNIQUE (напомним, что в расчете на каждую таблицу может быть задан только один первичный ключ).

После того как будет задано ограничение UNIQUE, должно соблюдаться такое условие, чтобы все значения в столбцах, указанных в этом ограничении, были уникальными. При обнаружении попытки обновить или вставить строку со значением, которое уже имеется в столбце с ограничением уникальности, СУБД SQL Server вырабатывает сообщение об ошибке и отклоняет попытку ввести такую строку.

> В отличие от ограничения первичного ключа, применение ограничения UNIQUE не приводит к тому, что автоматически исключается возможность вставить в соответствующий столбец NULL-значение. В данном случае возможность применения NULL-значений зависит от того, как задана опция NULL для указанного столбца таблицы. Но следует учитывать, что даже если вставка NULL-значений разрешена, фактически допускается вставка в столбец только одного такого значения (хотя одно NULL-значение не равно другому NULL-значению, с точки зрения применения ограничения UNIQUE эти значения все равно рассматриваются как дубликаты).

Итак, к описанию данного ограничения больше нельзя что-либо добавить (к тому же при его изучении применим основной объем сведений, касающихся первичных ключей), поэтому приступим непосредственно к изучению примеров кода. Для этого создадим еще одну таблицу в базе данных Accounting, но на этот раз назовем ее Shippers:

```
CREATE TABLE Shippers
(
   ShipperID      int         IDENTITY    NOT NULL
      PRIMARY KEY,
   ShipperName    varchar(30)             NOT NULL,
   Address        varchar(30)             NOT NULL,
   City           varchar(25)             NOT NULL,
   State          char(2)                 NOT NULL,
   Zip            varchar(10)             NOT NULL,
   PhoneNo        varchar(14)             NOT NULL
      UNIQUE
```

После этого необходимо вызвать на выполнение процедуру sp_helpconstraint применительно к таблице Shippers и проверить, действительно ли таблица Shippers была создана с учетом требуемых ограничений.

Создание ограничений UNIQUE на существующих таблицах

Как уже было сказано, ограничение UNIQUE действует в основном по таким же принципам, как ограничения первичных и внешних ключей. Рассмотрим следующий пример создания ограничения UNIQUE на таблице Employees:

```
ALTER TABLE Employees
   ADD CONSTRAINT AK_EmployeeSSN
   UNIQUE (SSN)
```

Сразу же после применения процедуры sp_helpconstraint обнаруживается, что ограничение создано в соответствии с запланированным действием и правильно указаны столбцы, относящиеся к ограничению.

Отметим, что в приведенном выше примере аббревиатура AK, используемая в имени ограничения, расшифровывается как Alternate Key (альтернативный ключ). С другой стороны, для обозначения первичных ключей и внешних ключей применяются аббревиатуры PK (Primary Key) и FK (Foreign Key). А в именах ограничений UNIQUE часто используется префикс UQ, или просто U.

Ограничения CHECK

Удобным свойством ограничений CHECK является то, что эти ограничения не обязательно должны применяться только к какому-то конкретному столбцу. Безусловно, указанные ограничения могут относиться лишь к некоторому столбцу, но также допускается их распространение по существу на всю таблицу, в том смысле, что с их помощью может осуществляться проверка значений в одном столбце на основании значений другого столбца (при условии, что все эти столбцы принадлежат к одной и той же таблице, а значения берутся из одной и той же обновляемой или вставляемой строки). С помощью ограничений CHECK может также осуществляться проверка того, соответствует ли некоторое сочетание значений столбцов заданному критерию.

Ограничения CHECK определяются на основании таких же правил, которые распространяются на операции проверки, используемые в конструкции WHERE. Примеры критериев, которые могут применяться в ограничении CHECK, приведены в табл. 6.3.

Таблица 6.3. Примеры применения ограничения CHECK

Назначение	Код SQL
Обеспечение применения в столбце Month только допустимых значений	`BETWEEN 1 AND 12`
Правильное форматирование номера карточки социального обеспечения	`LIKE '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]'`
Регламентация списка допустимых значений в поле Shippers	`IN ('UPS', 'Fed Ex', 'USPS')`
Регламентация применения только положительных значений цены	`UnitPrice >= 0`
Ссылка на другое поле в той же строке	`ShipDate >= OrderDate`

В действительности в табл. 6.3 приведена лишь незначительная часть всех возможных примеров, поскольку количество вариантов применения различных операций сравнения в конструкции CHECK является практически бесконечным. В ограничении CHECK может быть задано почти любое такое же выражение, которое допускается задавать в конструкции WHERE. К тому же ограничения CHECK позволяют достичь гораздо более высокой производительности по сравнению с альтернативными средствами проверки допустимости данных (правилами и триггерами).

Продолжая ряд примеров, основанных на использовании базы данных Accounting, внесем в таблицу Customers изменение, позволяющее проверять допустимость даты в поле DateInSystem (это — поле системной даты, в котором не может находиться значение будущей даты):

```
ALTER TABLE Customers
   ADD CONSTRAINT CN_CustomerDateInSystem
   CHECK
   (DateInSystem <= GETDATE ())
```

Теперь попытаемся выполнить вставку строки со значением, нарушающим ограничение CHECK; эта попытка должна привести к возникновению ошибки:

```
INSERT INTO Customers
   (CustomerName, Address1, Address2, City, State, Zip, Contact,
    Phone, FedIDNo, DateInSystem)
VALUES
   ('Customer1', 'Address1', 'Add2', 'MyCity', 'NY', '55555',
    'No Contact', '553-1212', '930984954', '12-31-2049')
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint
"CN_CustomerDateInSystem". The conflict occurred in database "Accounting",
table "dbo.Customers", column 'DateInSystem'.
The statement has been terminated.
```

Если после этого будет внесено такое исправление, чтобы данные, вводимые в столбец DateInSystem, соответствовали критерию, заданному в ограничении CHECK (относились к дате, совпадающей с нынешней или предшествующей ей), то попытка выполнить оператор INSERT завершится успешно.

Ограничения DEFAULT

Ограничения DEFAULT принадлежат к одному из двух различных типов инструментальных средств обеспечения целостности данных, которые можно рассматривать как относящиеся к "заданным по умолчанию значениям". К сожалению, в связи с наличием двух таких разных средств почти с одинаковыми названиями возникает значительная путаница, но автор приложит все свои усилия, чтобы внести ясность (и надеется, что это ему удастся).

Другими типами конструкций, применяемыми по умолчанию, являются правила и заданные по умолчанию значения, которые описаны ниже в данной главе.

Ограничения DEFAULT, как и все прочие типы ограничений, предусмотрены в синтаксической структуре определения таблицы. Ограничения DEFAULT указывают, какие действия должны быть выполнены, если происходит вставка новой строки, не

содержащей данных, соответствующих тому столбцу, к которому относится это ограничение. Вообще говоря, действием обычно является подстановка литерального значения (если в определении ограничения, скажем, задано применяемое по умолчанию значение зарплаты, равное нулю, либо указано значение "UNKNOWN" для столбца со строковыми данными) или подстановка одного из нескольких значений, формируемых системой, таких как дата, формируемая с помощью функции GETDATE().

Основные особенности ограничений DEFAULT описаны ниже.

- Значения, подстановка которых должна быть выполнена по умолчанию с помощью конструкции DEFAULT, используются только в операторах INSERT, а в случае их определения в операторах UPDATE и DELETE игнорируются.
- Если в операторе INSERT для столбца с конструкцией DEFAULT задано какое-либо значение, то предусмотренное по умолчанию значение не используется.
- Если же для такого столбца значение не задано, то всегда используется значение, предусмотренное по умолчанию.

Следует подчеркнуть, что предусмотренные по умолчанию значения предназначены для использования только в операторах INSERT. Но практика показывает, что для многих начинающих разработчиков программ для СУБД SQL Server причина такого положения дел остается непонятной. Тем не менее такая организация работы вполне обоснована — в то время, как происходит вставка строки в таблицу, СУБД SQL Server может использовать только те значения столбцов, которые приведены в операторе INSERT (если таковые имеются), или значения, заданные по умолчанию. Если же для какого-либо столбца в операторе вставки не указано ни то ни другое, то СУБД SQL Server вставляет в соответствующий столбец таблицы NULL-значение (которое по существу рассматривается как неопределенное значение), а если в определении столбца имеется конструкция NOT NULL, то СУБД SQL Server отвергает попытку вставки этой строки. Тем не менее после выполнения оператора вставки строки в таблице уже имеется значение в столбце, для которого задана конструкция DEFAULT, а в операторе UPDATE содержатся новые значения. Если в операторе обновления не предусмотрено новое значение для столбца с конструкцией DEFAULT, то СУБД SQL Server оставляет неизменным то значение, которое уже находится в столбце.

С другой стороны, если для столбца с конструкцией DEFAULT в операторе обновления предусмотрено значение, то также нет смысла использовать значение, заданное по умолчанию, поскольку применяется предоставленное значение.

Итак, заданное по умолчанию значение всегда используется в тех случаях, когда в операторе вставки данных не задано значение. Таким образом, описанные выше принципы организации работы при использовании конструкции DEFAULT являются достаточно простыми, но остается невыясненным еще один вопрос. В определенных обстоятельствах может потребоваться фактически ввести в столбец с конструкцией DEFAULT не значение, заданное по умолчанию, а NULL-значение. В данном случае является неприменимым вариант, в котором не задается значение в операторе вставки для столбца с конструкцией DEFAULT, поскольку вместо NULL-значения в него будет введено значение, предусмотренное по умолчанию. Поэтому если требуется ввести в такой столбец NULL-значение, то необходимо явно задать NULL в применяемом операторе INSERT.

Следует также отметить, что из правила о том, что при использовании оператора UPDATE не применяются значения, заданные по умолчанию, есть исключение. Такое исключение возникает при явном указании на то, что в операторе обновления должно использоваться значение, заданное по умолчанию. Для подобного указания необходимо ввести ключевое слово DEFAULT в качестве значения, которое должно появиться в обновляемом столбце.

Применение ограничения DEFAULT в операторе CREATE TABLE

Синтаксическая структура оператора создания таблицы с ограничением DEFAULT во многом напоминает структуру этого оператора со всеми другими ограничениями столбцов, которые рассматривались выше в данной главе. Для ввода в действие ограничения DEFAULT достаточно ввести его в конце определения столбца.

Рассмотрим пример применения ограничения DEFAULT. Для этого необходимо вначале удалить существующую таблицу Shippers, которая была создана ранее в этой главе. На этот раз будет создана более простая версия этой таблицы, включающая значение, применяемое по умолчанию:

```
CREATE TABLE Shippers
(
  ShipperID      int         IDENTITY    NOT NULL
    PRIMARY KEY,
  ShipperName    varchar(30)             NOT NULL,
  DateInSystem   smalldatetime           NOT NULL
    DEFAULT GETDATE ()
)
```

После выполнения сценария с этим оператором CREATE можно снова воспользоваться процедурой sp_helpconstraint, чтобы ознакомиться с полученными результатами. После этого для проверки того, как происходит подстановка значения, заданного по умолчанию, выполним вставку новой записи:

```
INSERT INTO Shippers
  (ShipperName)
VALUES
  ('United Parcel Service')
```

Затем выполним оператор SELECT применительно к таблице Shippers:

```
SELECT * FROM Shippers
```

Как показано ниже, для столбца DateInSystem вырабатывается значение, применяемое по умолчанию, поскольку в операторе вставки не было явно задано значение.

```
ShipperID        ShipperName              DateInSystem
---------------------------------------   --------------------------
1                United Parcel Service    2000-07-13 23:26:00

(1 row(s) affected)
```

Добавление ограничения DEFAULT к существующей таблице

Безусловно, конструкция DEFAULT, применяемая для добавления ограничения DEFAULT к существующей таблице, остается такой же, как и при создании таблицы, но в самом операторе модификации таблицы имеется небольшой нюанс. Оператор ALTER и конструкция ADD, предназначенная для добавления ограничения, остаются такими же, как и в других описанных выше операторах, но дополнительно предусматривается ключевое слово FOR, которое указывает СУБД SQL Server, какой столбец является целевым для ограничения DEFAULT. Пример такого оператора модификации таблицы приведен ниже.

```
ALTER TABLE Customers
   ADD CONSTRAINT CN_CustomerDefaultDateInSystem
      DEFAULT GETDATE() FOR DateInSystem
```

А следующий пример показывает, как задать литеральное значение в качестве применяемого по умолчанию:

```
ALTER TABLE Customers
   ADD CONSTRAINT CN_CustomerAddress
      DEFAULT 'UNKNOWN' FOR Address1
```

В любой таблице может быть задано несколько ограничений DEFAULT, как и всех прочих ограничений, кроме PRIMARY KEY.

> В определении таблицы могут применяться любые комбинации и сочетания описанных выше ограничений, но с учетом того, чтобы создаваемые ограничения не содержали взаимоисключающих условий. Например, нельзя допускать, чтобы в одном ограничении было задано условие col1 > col2, а в другом — условие col2 > col1. СУБД SQL Server допускает ввод подобных противоречащих друг другу условий, и проблема не обнаруживается до тех пор, пока не наступит этап прогона приложения.

Отмена действия ограничений

Иногда возникает необходимость отменить действие ограничения на время или навсегда. Вполне очевидно, что в СУБД SQL Server должны быть предусмотрены определенные способы удаления ограничений, но эта СУБД позволяет также просто перевести в неактивное состояние ограничение FOREIGN KEY или CHECK, оставив неизменным само определение этого ограничения.

На первый взгляд кажется, что отмена действия правила обеспечения целостности данных лишена смысла. И действительно, непонятно, для чего может потребоваться отменять действие механизма, позволяющего исключить появление в базе данных неправильных данных. Но обычно такая необходимость действительно возникает в той ситуации, когда в базе данных уже имеются неправильные данные. Такие данные подразделяются в основном на две описанные ниже категории.

- Данные, которые уже находились в базе данных ко времени создания ограничений.

- Данные, которые по какой-то причине необходимо ввести после того, как ограничение уже введено в действие.

> Ограничение PRIMARY KEY или UNIQUE нельзя отменять.

Игнорирование неправильных данных при создании ограничения

Синтаксическая структура всех операторов, с помощью которых могут быть заданы ограничения, в основном рассчитана на использование в тех обстоятельствах, когда ограничения создаются одновременно с тем, как создается таблица. Тем не менее достаточно часто возникают такие ситуации, что уже после ввода в действие приложения формулируются новые правила работы с данными. Подобное положение дел может быть обусловлено, например, тем, что во время проектирования базы данных не были учтены какие-то требования, из-за чего, допустим, в таблице Invoicing появились некоторые строки, содержащие отрицательное значение суммы оплаты по счету-фактуре. В связи с этим возникает необходимость добавить правило, не позволяющее вводить в базу данных счета-фактуры с отрицательными значениями суммы, но вместе с тем оставить неизменными существующие строки.

Чтобы ввести в действие новое ограничение, но исключить его применение к существующим данным, можно включить опцию WITH NOCHECK при подготовке оператора ALTER TABLE, предназначенного для добавления нового ограничения. Как обычно, рассмотрим применение этой опции на примере.

Таблица Customers, созданная в базе данных Accounting, имеет столбец Phone. Столбец Phone был создан с типом данных char, поскольку предполагалось, что все номера телефонов будут иметь одинаковую длину. Кроме того, для столбца Phone было задано значение длины 15, чтобы обеспечить наличие места, достаточного для размещения в поле Phone всех необходимых форматирующих символов. Тем не менее в объявлении таблицы Customers ничего не было предусмотрено для обеспечения того, чтобы строки, вставляемые в таблицу базы данных, действительно соответствовали требуемым критериям форматирования. Чтобы убедиться в этом, вставим строку со значением номера телефона в формате, не соответствующем ожиданиям, но допустим при этом такую ошибку, которая действительно может встретиться на практике при вводе номера телефона:

```
INSERT INTO Customers
   (CustomerName,
    Address1,
    Address2,
    City,
    State,
    Zip,
    Contact,
    Phone,
    FedIDNo,
    DateInSystem)
VALUES
   ('MyCust',
    '123 Anywhere',
```

```
    '',
    'Reno',
    'NV',
    80808,
    'Joe Bob',
    '555-1212',
    '931234567',
    GETDATE ())
```

Теперь введем в действие ограничение, позволяющее управлять форматированием поля Phone:

```
ALTER TABLE Customers
  ADD CONSTRAINT CN_CustomerPhoneNo
  CHECK
 (Phone LIKE '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

Но после вызова этого оператора на выполнение появляется следующее сообщение об ошибке:

```
Msg 547, Level 16, State 1, Line 1
ALTER TABLE statement conflicted with COLUMN CHECK constraint 'CN_
CustomerPhoneNo'. The conflict occurred in database 'Accounting', table
'Customers', column 'Phone'.
```

СУБД SQL Server вводит ограничение в действие только при том условии, что существующие данные соответствуют критериям, заданным в ограничении. В связи с этим для ввода ограничения в действие необходимо либо исправить существующие данные, либо ввести опцию WITH NOCHECK в оператор ALTER. Для этого достаточно дополнительно указать ключевое слово WITH NOCHECK в операторе ALTER, как показано ниже.

```
ALTER TABLE Customers
  WITH NOCHECK
  ADD CONSTRAINT CN_CustomerPhoneNo
  CHECK
  (Phone LIKE '([0-9][0-9][0-9]) [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

Если после этого будет снова вызван на выполнение тот же оператор INSERT, введенное ограничение проявит свое действие и попытка ввода данных будет отвергнута (напомним, что в прошлый раз этот оператор вставки был выполнен беспрепятственно):

```
Msg 547, Level 16, State 0, Line 1
The ALTER TABLE statement conflicted with the CHECK constraint
"CN_CustomerPhoneNo". The conflict occurred in database "Accounting", table
"dbo.Customers", column 'Phone'.
```

После того как оператор INSERT будет исправлен с учетом введенного ограничения, а затем снова вызван на выполнение, операция вставки строки завершится успешно:

```
INSERT INTO Customers
  (CustomerName,
   Address1,
   Address2,
   City,
```

```
    State,
    Zip,
    Contact,
    Phone,
    FedIDNo,
    DateInSystem)
VALUES
   ('MyCust',
    '123 Anywhere',
    '',
    'Reno',
    'NV',
    80808,
    'Joe Bob',
    '(800)555-1212',
    '931234567',
    GETDATE ())
```

Теперь выполним оператор SELECT применительно к таблице Customers. Приведенные ниже результаты показывают, что в столбце Phone этой таблицы имеются данные и соответствующие, и несоответствующие критериям ограничения CHECK.

```
SELECT CustomerNo, CustomerName, Phone FROM Customers
CustomerNo       CustomerName                Phone
-----------      --------------------        --------------
1                Billy Bob's Shoes           (360) 555-1234
2                Customer1                   553-1212
3                MyCust                      555-1212
5                MyCust                      (800) 555-1212

(2 row(s) affected)
```

Введенные ранее данные сохранены, поскольку отсутствует информация, позволяющая внести в них уточнения, а ввод новых данных осуществляется только после проверки их соответствия вновь заданным критериям.

Временная отмена существующего ограничения

Выше было описано, в связи с чем может потребоваться, чтобы при вводе в действие новых ограничений не проверялись существующие данные, а в этом разделе будет показано, из-за чего может возникнуть необходимость временной отмены существующего ограничения. Прежде всего следует отметить, что ввод в БД таких данных, в отношении которых заведомо известно, что они неправильны, может быть вызван той же причиной, по которой используется опция WITH NOCHECK — в базу данных требуется ввести существующие данные.

Существующими данными могут оказаться не только те данные, которые были введены ранее в базу данных. К такой категории могут также относиться данные, импортируемые из эксплуатировавшейся ранее базы данных или из какой-то другой системы. Но так или иначе, при вводе этих данных возникает та же проблема — имеются некоторые существующие данные, которые не соответствуют правилам, заданным с помощью ограничений, но эти данные все равно должны быть введены в таблицу.

Безусловно, одним из способов решения этой задачи может стать уничтожение ограничения, ввод требуемых данных, а затем повторное задание того же ограни-

чения с использованием опции WITH NOCHECK. Но это — слишком сложный способ! К счастью, можно обойтись без выполнения всех этих действий. Вместо этого достаточно выполнить оператор ALTER с опцией NOCHECK, который отменяет действие рассматриваемого ограничения. Ниже приведен пример, в котором показано, как отменить ограничение CHECK, введенное в действие в одном из операторов из предыдущего раздела.

```
ALTER TABLE Customers
   NOCHECK
   CONSTRAINT CN_CustomerPhoneNo
```

После этого можно снова выполнить тот же оператор INSERT, который, как уже было проверено ранее, отвергается СУБД SQL Server, если ограничение введено в действие:

```
INSERT INTO Customers
   (CustomerName,
    Address1,
    Address2,
    City,
    State,
    Zip,
    Contact,
    Phone,
    FedIDNo,
    DateInSystem)
VALUES
   ('MyCust',
    '123 Anywhere',
    '',
    'Reno',
    'NV',
    80808,
    'Joe Bob',
    '555-1212',
    '931234567',
    GETDATE())
```

На этот раз с помощью оператора INSERT удается ввести в таблицу данные, не соответствующие условиям ограничения.

В связи с указанной возможностью отменять ограничения представляет также интерес вопрос о том, как узнать, отменено ли действие какого-то конкретного ограничения, или нет. Способ такой проверки, основанный на том, что подготавливается фиктивная строка и предпринимается попытка вставить ее в таблицу, чтобы проверить, действует ли заданное ограничение, был бы слишком трудоемким. К счастью, в СУБД SQL Server предусмотрена процедура, позволяющая определить состояние ограничения (а также, разумеется, много других процедур, позволяющих находить ответы на большинство, но не все подобные вопросы). В данном случае достаточно применить уже знакомую нам процедуру sp_helpconstraint. Оператор, с помощью которого можно использовать эту процедуру применительно к таблице Customers, является несложным:

```
EXEC sp_helpconstraint Customers
```

Объем результатов, предоставляемых этой процедурой, слишком велик для того, чтобы эти результаты можно было привести на страницах этой книги, но отметим, что во втором результирующем наборе, вырабатываемом этой процедурой, имеется столбец status_enabled. Приведенная в указанном столбце информация позволяет узнать состояние каждого ограничения, а в данном случае для рассматриваемого ограничения должно быть указано состояние Disabled (Отменено).

После того как появится возможность снова ввести в действие требуемое ограничение, можно просто разрешить его использование, выполнив такой же оператор ALTER, но с опцией CHECK вместо опции NOCHECK:

```
ALTER TABLE Customers
   CHECK
   CONSTRAINT CN_CustomerPhoneNo
```

После вызова на выполнение того же оператора INSERT для проверки функционирования ограничения появится такое же сообщение об ошибке, как и раньше:

```
Msg 547, Level 16, State 0, Line 1
The INSERT statement conflicted with the CHECK constraint
"CN_CustomerDateInSystem". The conflict occurred in database "Accounting",
table "dbo.Customers", column 'DateInSystem'.
The statement has been terminated.
```

Безусловно, и в данном случае рекомендуется снова вызвать на выполнение процедуру sp_helpconstraint и ознакомиться с содержимым столбца status_enabled. Если в этом столбце указано состояние Enabled (Разрешено), то рассматриваемое ограничение снова должно действовать.

Конструкции, подобные ограничениям, правила и значения, применяемые по умолчанию

Такие конструкции, как **правила** и **заданные по умолчанию значения**, были предусмотрены в СУБД SQL Server намного раньше, чем ограничения CHECK и DEFAULT. В определенном смысле эти конструкции можно рассматривать как унаследованные от прежних версий SQL Server, но они не лишены определенных преимуществ.

Несмотря на сказанное, нет смысла подробно описывать эти конструкции, и остается лишь порекомендовать ознакомиться с ними в такой степени, чтобы узнать требования к обратной совместимости и получить возможность работать с унаследованным кодом. Правила и заданные по умолчанию значения несовместимы со стандартом ANSI (в связи с чем возникают проблемы переносимости), к тому же они не позволяют достичь такой же высокой производительности, как при использовании ограничений. Корпорация Microsoft перевела правила и заданные по умолчанию значения в категорию программных конструкций, поддерживаемых только для обеспечения обратной совместимости, начиная с версии 7.0. С тех пор прошло больше шести лет и выпущены три новые версии. Такое положение дел вряд ли может внушить оптимизм тем разработчикам, которые задумываются над тем, продолжится ли поддержка этих средств в будущем. Сам автор не зашел бы настолько далеко, чтобы предложить вам немедленно приступить к просмотру и замене любого кода с устаревшими конструкциями, который может встретиться в вашей работе, но, безусловно, в любом вновь разрабатываемом коде необходимо использовать ограничения.

Правила и заданные по умолчанию значения отличаются от ограничений прежде всего тем, как организовано их применение. Дело в том, что ограничения представляют собой свойства самой таблицы и не существуют как отдельно взятые конструкции, тогда как правила и заданные по умолчанию значения являются действительными объектами, существующими отдельно от других объектов. Ограничения определяются в составе объявления таблицы, а правила и заданные по умолчанию значения объявляются независимо, после чего выполняется их "привязка" к таблице.

Таким образом, правила и заданные по умолчанию значения являются независимыми объектами, что дает возможность использовать их повторно, без неоднократного переопределения. В действительности область применения правил и заданных по умолчанию значений не ограничивается их привязкой только к таблицам; допускается также привязка этих объектов к типам данных, что существенно расширяет возможности по созданию определяемых пользователем типов данных, обладающих широкими функциональными возможностями.

Правила

Правила весьма напоминают ограничения CHECK. Правила имеют свойства, аналогичные описанным выше свойствам ограничений CHECK, за единственным исключением, — область применения правил ограничена тем, что они позволяют работать одновременно только с одним столбцом. Одно и то же правило можно связать отдельно с несколькими столбцами в таблице, но это правило будет действовать независимо применительно к каждому столбцу, поскольку в нем вообще не учитывается наличие других столбцов. Это означает, что ограничение, определенное как (QtyShipped <= QtyOrdered), не может применяться в качестве правила (в нем упоминается больше чем один столбец), а ограничение LIKE ([0-9][0-9][0-9]) может использоваться в этом качестве (оно относится только к тому столбцу, с которым может быть также связано аналогичное правило).

Вначале определим правило, чтобы можно было проще показать его отличие от ограничения:

```
CREATE RULE SalaryRule
  AS @Salary > 0
```

Здесь заслуживает внимания то, что сравниваемое с конкретным числом значение представлено в виде переменной. В качестве этого значения могут быть взяты данные из контролируемого столбца, после чего осуществляется подстановка такого значения вместо переменной @Salary. Таким образом, в данном примере правило показывает, что столбец, к которому оно будет привязано, должен содержать значения больше нуля.

После создания данного правила можно проверить, как оно представлено в базе данных. Для этого можно воспользоваться процедурой sp_helptext:

```
EXEC sp_helptext SalaryRule
```

После вызова этого оператора на выполнение отображается точное определение правила:

```
Text
----------------------------------
CREATE RULE SalaryRule
     AS @Salary > 0
```

Непосредственно после его определения правило не выполняет никаких действий. При попытке вставить строку в таблицу Employees обнаруживается, что в текущих обстоятельствах допускается ввод любого значения без учета каких-либо ограничений, кроме обусловленных применяемым типом данных.

Для активизации правила необходимо воспользоваться специальной хранимой процедурой sp_bindrule. Предположим, что требуется выполнить привязку правила SalaryRule к столбцу Salary таблицы Employees. Для этого применяется следующий синтаксис:

```
sp_bindrule <'rule'>, <'object_name'>, [<'futureonly_flag'>]
```

Часть <'rule'> этого определения является достаточно простой — в ней должно быть указано правило, подлежащее привязке. Часть <'object_name'> также не представляет никаких сложностей — она указывает на объект (столбец или определяемый пользователем тип данных), к которому следует выполнить привязку правила. Определенные нюансы следует учитывать только при использовании параметра <'futureonly_flag'>, который применяется лишь в случае привязки правила к определяемому пользователю типу данных. По умолчанию этот параметр имеет значение off. Но если при вызове процедуры sp_bindrule <'rule'> в качестве него будет задано True или 1, то связывание правила будет осуществляться только применительно к новым столбцам, для которых задается соответствующий тип данных, определяемый пользователем. Во всех столбцах, для которых уже был определен тот же тип данных в старой форме, будет по-прежнему использоваться форма, для которой еще не было задано правило.

В рассматриваемом случае происходит просто привязка правила к столбцу, поэтому согласно синтаксическому определению требуются только первые два параметра:

```
sp_bindrule 'SalaryRule', 'Employees.Salary'
```

В данном примере интерес представляет параметр, заданный в качестве object_name, — в нем указаны и имя таблицы Employees, и имя столбца Salary, разделенные точкой (.). Причина применения такой конструкции заключается в том, что правило не относится к какой-либо конкретной таблице до тех пор, пока не будет выполнена его привязка, поэтому необходимо указывать и таблицу, и столбец, к которому должно быть привязано это правило. Если не используется структура именования tablename.column, то СУБД SQL Server действует согласно предположению, что указанное имя относится к определяемому пользователем типу данных, а если таковое не обнаруживается, то появляется довольно непонятное сообщение об ошибке, относящееся к той ситуации, как будто была предпринята попытка привязки правила к типу данных:

```
Msg 15148, Level 16, State 1, Procedure sp_bindrule, Line 190
The data type or table column 'Salary' does not exist or you do not have permission.
```

После успешного осуществления привязки правила к столбцу попытка вставки или обновления строки таблицы Employees с отрицательным значением в столбце Salary приводит к нарушению правила и появлению сообщения об ошибке. Чтобы отменить привязку правила к столбцу, можно воспользоваться процедурой sp_unbindrule:

```
EXEC sp_unbindrule 'Employees.Salary'
```

Синтаксическая структура процедуры sp_unbindrule также предусматривает возможность применения параметра futureonly_flag, причем, как и в случае проце-

дуры sp_bindrule, этот параметр не используется применительно к определениям столбцов. Если процедура sp_unbindrule вызывается на выполнение с параметром futureonly_flag, которому присвоено значение on, а объектом действия процедуры является тип данных, определяемый пользователем (а не конкретный столбец), то отмена привязки распространяется только на те случаи, в которых этот тип данных будет использоваться в дальнейшем, а в существующих столбцах с указанным типом данных по-прежнему будет применяться то же правило.

Удаление правила

Если из базы данных необходимо полностью удалить определение некоторого правила, то можно воспользоваться оператором DROP, применение которого было показано выше на примере таблиц:

```
DROP RULE <rule name>
```

Заданные по умолчанию значения

Заданные по умолчанию значения еще в большей степени напоминают свои аналоги, ограничения DEFAULT, чем правила напоминают ограничения CHECK. И действительно, заданные по умолчанию значения и ограничения DEFAULT действуют полностью одинаково, а единственное реальное различие между ними заключается в том, что они по-разному закрепляются за таблицей. Кроме того, заданные по умолчанию значения (которые в отличие от ограничений представляют собой объекты базы данных) поддерживают определяемые пользователем типы данных.

> Для большинства начинающих разработчиков бывает чрезвычайно трудно понять, чем отличаются друг от друга заданные по умолчанию значения и ограничения DEFAULT. Особую путаницу вызывает то, что и те и другие конструкции имеют почти одинаковое назначение. А когда речь идет о "заданном по умолчанию значении", непонятно, подразумеваются ли под этим объекты базы данных, применяемые в качестве заданных по умолчанию значений (которым посвящен настоящий раздел), или кратко обозначается значение, которое действительно используется по умолчанию (в том случае, если явно не задано какое-либо действительное значение). С другой стороны, понятие "ограничение DEFAULT" относится к той конструкции, которая не основана на использовании отдельного объекта базы данных, поскольку эта конструкция является неотъемлемой частью определения таблицы.

Синтаксис определения заданного по умолчанию значения во многом напоминает синтаксис правила:

```
CREATE DEFAULT <default name>
AS <default value>
```

Рассмотрим пример определения заданного по умолчанию значения, равного нулю, для столбца Salary:

```
CREATE DEFAULT SalaryDefault
  AS 0
```

И в этом случае заданное по умолчанию значение, не привязанное к какому-либо объекту, не выполняет реальных функций. Для привязки заданных по умолчанию значений используется процедура sp_bindefault, имеющая синтаксис, идентичный

синтаксису процедуры sp_bindrule (эти процедуры различаются только своими именами):

```
EXEC sp_bindefault 'SalaryDefault', 'Employees.Salary'
```

А для отмены привязки заданного по умолчанию значения к столбцу применяется процедура sp_unbindefault:

```
EXEC sp_unbindefault 'Employees.Salary'
```

Следует учитывать, что параметр futureonly_flag допускается применять также в вызове хранимых процедур sp_bindefault и sp_unbindefault, но этот параметр также не используется применительно к столбцам.

Удаление заданных по умолчанию значений

Если из базы данных необходимо полностью удалить как объект заданное по умолчанию значение, то можно воспользоваться оператором DROP, применение которого уже было показано по отношению к таблицам и правилам:

```
DROP DEFAULT <default name>
```

Определение того, в каких таблицах и типах данных используются те или другие правила либо заданные по умолчанию значения

Если в процессе эксплуатации базы данных возникает необходимость удалить или модифицировать правила или заданные по умолчанию значения, целесообразно вначале ознакомиться с тем, в каких таблицах и типах данных они используются. И для этого можно воспользоваться соответствующей системной хранимой процедурой, предусмотренной в СУБД SQL Server. Эта процедура называется sp_depends и имеет следующий синтаксис:

```
EXEC sp_depends <object name>
```

Процедура sp_depends предоставляет список всех объектов, зависящих от того объекта, в отношении которого запрашивается информация.

К сожалению, нельзя полностью рассчитывать на получение с помощью процедуры sp_depends информации обо всех объектах, зависящих от указанного родительского объекта. Дело в том, что СУБД SQL Server поддерживает принцип организации работы, который иногда упоминается под названием "отложенного разрешения имен". По существу, под понятием отложенного разрешения имен подразумевается возможность создавать объекты (прежде всего, хранимые процедуры), которые зависят от другого объекта, причем еще до создания второго объекта (целевого объекта зависимости). Например, современные версии SQL Server допускают создание хранимой процедуры, которая ссылается на некоторую таблицу, еще до того, как будет создана сама указанная в ней таблица. В подобных случаях СУБД SQL Server не имеет возможности указать в списке, формируемом с помощью процедуры sp_depends, рассматриваемую таблицу как участвующую в интересующей нас зависимости. Поэтому в списке зависимостей, формируемом с помощью процедуры sp_depends, не будут приведены сведения о зависимости даже после создания соответствующей таблицы.

Применение триггеров для обеспечения целостности данных

Описанию триггеров посвящена отдельная глава настоящей книги, но любое описание ограничений, правил и заданных по умолчанию значений не будет полным, если в нем хотя бы не упоминаются триггеры.

Триггеры наиболее широко применяются для реализации правил обеспечения целостности данных. Триггерам посвящена одна из следующих глав книги, поэтому в данном разделе отметим лишь то, что с помощью триггеров осуществляется очень широкий набор действий, касающихся целостности данных. Эти действия обычно являются настолько сложными, что невозможно надеяться на их осуществление с помощью какого-либо ограничения или правила. Недостатком триггеров (а недостатки, как известно, можно найти во всем) является то, что они вносят существенные дополнительные издержки и поэтому практически в любых обстоятельствах весьма (и весьма) значительно замедляют работу. Триггеры по самой своей сути являются процедурными конструкциями (и именно этим обусловлены их широкие возможности), но их следует вводить в действие только после того, как процесс определения всех прочих объектов базы данных полностью заканчивается. Кроме того, к использованию триггеров следует прибегать только после того, как возможности применения всех других средств обеспечения целостности данных будут исчерпаны.

Выбор используемых средств обеспечения целостности данных

В предыдущих разделах данной главы описаны все виды средств обеспечения целостности данных, а в настоящем разделе приведена информация о том, какими соображениями следует руководствоваться при выборе средств, наиболее подходящих для использования в конкретной ситуации. Что касается ограничений, то некоторые из них имеют довольно четко обозначенную область применения (к ним относятся ограничения первичного и внешнего ключа, а также ограничения уникальности), поэтому речь может идти лишь о том, должны ли они использоваться в конкретной таблице или нет. С другой стороны, функции ограничений других типов и прочих средств обеспечения целостности данных в определенной степени перекрываются, поэтому задача выбора средств, наиболее подходящих в конкретной ситуации, может оказаться довольно сложной. Вообще говоря, в настоящей главе были приведены основные сведения о преимуществах и недостатках каждой из рассматриваемых конструкций, а в табл. 6.4 эти сведения кратко представлены вместе для удобства пользования.

Правила и заданные по умолчанию значения в основном используются, если в базе данных должна быть реализована довольно надежная логическая модель и предусмотрено широкое применение определяемых пользователем типов данных. В таком случае правила и заданные по умолчанию значения позволяют использовать широкий набор функциональных средств и упростить сопровождение базы данных, не требуя значительных затрат труда со стороны программистов. Однако, прибегая к применению правил и заданных по умолчанию значений, следует помнить, что в какой-то из будущих версий SQL Server поддержка этих конструкций может прекратиться. Возможно, этот день наступит не скоро, но обязательно наступит.

Таблица 6.4. Преимущества и недостатки некоторых средств обеспечения целостности данных

Средство поддержки целостности данных	Преимущества	Недостатки
Ограничения	Обеспечивают высокое быстродействие. Позволяют ссылаться на другие столбцы. Вызываются до начала выполнения обрабатывающей основной части оператора. Являются совместимыми со стандартом ANSI	Должны быть переопределены отдельно для каждой таблицы. Не позволяют ссылаться на другие таблицы. Не могут быть привязаны к типам данных
Правила и заданные по умолчанию значения	Являются независимыми объектами. Допускают повторное использование. Могут быть привязаны к типам данных. Вызываются до начала выполнения основной части оператора	Отличаются немного более низким быстродействием, чем ограничения. Не позволяют формировать ссылки с одних столбцов на другие. Не позволяют ссылаться на другие таблицы. В действительности предназначены только для обеспечения обратной совместимости!!!
Триггеры	Допускают исключительно широкое применение. Позволяют ссылаться на другие столбцы и другие таблицы. Обеспечивают даже возможность использовать инфраструктуру .NET для доступа к информации, внешней по отношению к СУБД SQL Server	Вызываются после начала выполнения обрабатывающей основной части оператора. Характеризуются высокими издержками

Триггеры должны использоваться только в том случае, если ограничения не позволяют достичь намеченной цели. Как и ограничения, триггеры закрепляются за таблицей и должны быть вновь переопределены после создания каждой новой таблицы. Значительным преимуществом триггеров является то, что с их помощью можно осуществить большинство действий по обеспечению целостности данных, которые могут когда-либо потребоваться. И действительно, в свое время триггеры лежали в основе общепринятого способа принудительной поддержки ограничений внешних ключей (до тех пор, пока не были введены ограничения внешних ключей как отдельные конструкции). Эта тема рассматривается более подробно в одной из следующих глав книги.

Таким образом, для обеспечения целостности данных следует преимущественно использовать ограничения. Эти конструкции являются быстродействующими, а задача их создания не так уж сложна. Недостаток ограничений состоит в том, что их функциональные возможности строго лимитированы (в частности, любые ограничения, кроме ограничений внешнего ключа, не позволяют ссылаться на другие таблицы). А если в базе данных широко применяются однотипные условия ограничений, то приходится заниматься однообразной и скучной работой — переопределять одни и те же ограничения снова и снова.

Независимо от того, какой механизм обеспечения целостности вводится в действие (ключи, триггеры, ограничения, правила, заданные по умолчанию значения), всегда следует учитывать рекомендацию, которую можно выразить одним словом — обоснованность.

Каждый новый объект, создаваемый в базе данных, вносит дополнительные издержки, поэтому, прежде чем ввести какое-либо дополнение в базу данных, необходимо тщательно взвесить, оправдано ли такое дополнение. Избегайте таких ошибок, как избыточные реализации средств обеспечения целостности данных (например, трудно даже представить себе, сколько раз автору приходилось сталкиваться с таким проектом базы данных, в котором для обеспечения ссылочной целостности были определены внешние ключи, и для решения той же задачи дополнительно использовались триггеры). Прежде чем ввести любые ограничения, проверьте, какие ограничения уже заданы, и убедитесь в том, что намеченные к вводу ограничения действительно позволят получить те результаты, на которые вы рассчитываете.

Резюме

В настоящей главе описаны механизмы обеспечения целостности данных различных типов, без которых невозможно создать бесперебойно функционирующую базу данных. По-видимому, одним из наибольших преимуществ реляционных СУБД является то, что они позволяют возложить ответственность за обеспечение целостности данных на базу данных, а не рассчитывать исключительно на безукоризненную работу приложения. Благодаря этому появляется возможность добиваться соблюдения правил обработки данных даже при выполнении произвольных запросов, а также разрабатывать многочисленные приложения на основе единого подхода к решению проблем целостности данных.

В следующих главах рассматривается связь между некоторыми формами ограничений и индексами, а также приведено описание расширенных правил обеспечения целостности данных, которые могут быть реализованы с использованием триггеров. Кроме того, будет показано, как зависит выбор возможных проектных решений от используемых механизмов поддержки целостности.

7

Дополнительные сведения о запросах

Приступив несколько лет тому назад к написанию своей первой книги, посвященной СУБД SQL Server, автор столкнулся с проблемой выбора места для описания более сложных запросов среди других тем, а когда начал работу над настоящей книгой, снова был вынужден решать ту же проблему. Дело в том, что, определяя способ компоновки материала для книги об SQL Server, приходится думать, о чем рассказать раньше, о "курице" или о "яйце", – описать вначале проблематику составления сценариев, выбора переменных и раскрыть тому подобные темы или сразу же приступить к описанию операций, с которыми относительно неопытные пользователи могут начать работу задолго до того, как приступят к написанию серверных сценариев. В конечном итоге было решено в первую очередь уделить больше внимания запросам.

До начала изучения понятий, изложенных в данной главе, читатель должен подготовиться к тому, что ему придется столкнуться с новым способом мышления, поскольку речь фактически пойдет об операциях с множествами. Определенное представление об этом вы уже могли получить, знакомясь с тем, что было сказано в предыдущих главах в отношении соединений, а в настоящей главе указанный подход еще больше расширяется и углубляется. При использовании подхода, основанного на процедурном программировании, обработка данных происходит совсем иначе, но даже если разработчик не привык организовывать свою работу исключительно на основе процедурного программирования, остается фактом неизменное свойство любого человека – разбивать сложные задачи на меньшие подзадачи (логические этапы, которые можно сравнить с подпрограммами), а не использовать комплексный подход и рассматривать все обрабатываемые данные как единое целое (как "множество"), руководствуясь тем подходом, который принят в языке SQL.

Безусловно, в версии SQL Server 2005 обеспечивается более широкая поддержка концепций процедурных языков, но автор призывает в первую очередь постараться

освоить комплексный подход к решению задач. Не следует думать, что для этого достаточно изучить один-два запроса. Тем не менее даже если такой подход не удается освоить сразу, часто возникает возможность вначале применить для решения задачи несколько меньших запросов, а затем постепенно соединить их в единый, более крупный запрос, позволяющий непосредственно решить всю задачу. Итак, попытайтесь в первую очередь найти решение всей задачи в целом, а если этого не удается сделать, не отступайте, разбивайте ее на меньшие подзадачи, а затем снова соединяйте этапы решения в единое целое, продолжая эту работу до тех пор, пока она остается оправданной.

Именно этот подход лежит в основе "нового способа мышления", который предлагает освоить автор, — стремление найти способ решения всей задачи в целом, а не разбивать решение на этапы. Составляя программы на большинстве языков программирования, обычно требуется действовать в линейной последовательности, а при использовании языка SQL чаще приходится мыслить в терминах теории множеств. Дело в том, что многие операции в языке SQL практически полностью соответствуют операциям теории множеств, таким как объединение множеств (`A union B`) или пересечение множеств (`A intersect B`). Программирование на языке SQL в большей степени требует поиска тех признаков в данных, которые бы позволили применить к ним операции комплексной обработки, чем подготовки этапов пошаговой обработки данных.

Таким образом, в основе операций, выполняемых с помощью языка SQL, лежит понятие общности атрибутов в множествах данных. В настоящей главе это понятие будет применяться для создания таких запросов, которые позволяют осуществлять выборку данных сразу по нескольким критериям. По существу, в этой главе рассматриваются такие способы выборки данных, для которых на первый взгляд требуется применить несколько запросов, но фактически все необходимые средства обработки данных будут помещены в такую конструкцию, которая будет действовать как единое целое. Кроме того, речь пойдет о том, какая производительность может быть достигнута при выполнении запросов, а также о том, как достичь максимально эффективного выполнения запросов.

Ниже перечислены темы, которые будут рассматриваться в настоящей главе.

- Вложенные подзапросы.
- Связанные подзапросы.
- Производные таблицы.
- Использование операции `EXISTS`.
- Оптимизация производительности запросов.

Кроме того, в этой главе будет показано, как можно с помощью подзапросов решать задачи, которые на первый взгляд кажутся полностью неподдающимися решению, и насколько может измениться производительность запросов из-за небольших корректировок в отдельных конструкциях запросов, которые с виду кажутся незначительными.

Общее определение понятия подзапроса

Подзапрос — это обычный запрос на языке SQL, который вложен в другой запрос. Подзапрос представляет собой оператор `SELECT`, который применяется для выборки определенной части данных или служит в качестве условия для другого запроса. Текст подзапроса должен быть заключен в круглые скобки.

Как правило, подзапросы используются для выполнения одной из описанных ниже функций.

- Разбивка запроса на последовательность логических этапов.
- Составление списка, который вкладывается в конструкцию WHERE с помощью ключевого слова IN, EXISTS, ANY или ALL.
- Создание поисковой таблицы, соответствующей каждой отдельной строке в родительском запросе.

Чаще всего подзапрос можно разработать и создать очень легко, но иногда задача составления подзапроса становится чрезвычайно трудоемкой. Как правило, степень сложности задачи подготовки подзапроса зависит от того, насколько нетривиальной является связь между внутренним запросом (подзапросом) и внешним запросом (главным запросом).

Следует также отметить, что вместо большинства операторов с подзапросами (но это определенно не относится ко всем таким операторам) вполне возможно применить операторы с соединениями. Причем в тех случаях, когда действительно можно использовать соединения, именно такой способ организации работы является предпочтительным.

Автору однажды пришлось вступить в довольно продолжительный спор со своим сотрудником (при этом в течение нескольких дней произошел обмен почти что тремя десятками писем по электронной почте с примерами, доводами и т.д.). Этот спор возник по поводу того, какими сравнительными преимуществами и недостатками обладают соединения и подзапросы.

В соответствии с традициями, сложившимися в среде пользователей языка SQL, при любой возможности следует всегда использовать соединения, и я отстаивал именно эту точку зрения (исходя из своего опыта, а не сложившихся традиций; читатель уже, вероятно, заметил, что в некоторых местах данной книги была подчеркнута мысль, что традиционные взгляды могут оказаться ошибочными). С другой стороны, мой сотрудник стремился доказать, что фактически для выполнения подзапросов требуется меньше издержек. В конечном итоге было решено проверить обоснованность тех и других утверждений на практике.

Как и следовало ожидать, полученные результаты показали, что в различных обстоятельствах становится оправданным либо то, либо другое мнение. Более подробно о том, каковыми являются эти обстоятельства, будет сказано ближе к концу данной главы, после изложения всех необходимых для этого сведений.

Итак, после теоретического знакомства с тем, что представляют собой подзапросы, рассмотрим некоторые конкретные типы подзапросов и примеры их применения.

Создание вложенных подзапросов

Вложенным подзапросом является такой подзапрос, который выполняет исключительно однонаправленное действие — возвращает либо единственное значение, предназначенное для использования во внешнем запросе, либо, возможно, полный список значений, которые предназначены для использования в операции IN. В том случае, если требуется обеспечить осуществление явно заданной операции сравнения на равенство, "=", приходится применять такой подзапрос, который возвращает единственное значение, иными словами, значение одного поля одной строки. Если же предполагается, что подзапрос должен возвратить список значений, то в сочетании с внешним запросом должна использоваться операция IN.

Не вдаваясь в подробности, можно отметить, что синтаксическая конструкция запроса с вложенным подзапросом должна выглядеть примерно так, как показано в двух следующих примерах синтаксиса:

```
SELECT <SELECT list>
FROM <SomeTable>
WHERE <SomeColumn> = (
        SELECT <single column>
        FROM <SomeTable>
        WHERE <condition that results in only one row returned>)
```

или

```
SELECT <SELECT list>
FROM <SomeTable>
WHERE <SomeColumn> IN (
        SELECT <single column>
        FROM <SomeTable>
        [WHERE <condition>)]
```

Безусловно, синтаксическая конструкция практически применяемых операторов может выглядеть немного иначе. Это связано не только с тем, что могут изменяться списки выборки и уточняться имена таблиц, но и может быть также обусловлено тем, что во внешнем или внутреннем запросе, а также в обоих запросах может применяться соединение нескольких таблиц.

Запросы с вложенными подзапросами, в которых используются операторы SELECT, возвращающие единственные значения

Приступим к изучению нюансов использования подзапросов на конкретном примере. Предположим, например, что требуется определить идентификаторы ProductID всех товаров, проданных в первый день после того, как любой товар был куплен в данной торговой системе.

Если уже известно, каковым является тот первый день, когда в системе был введен заказ, то проблема не возникает; достаточно применить запрос, который выглядит примерно таким образом:

```
SELECT DISTINCT o.OrderDate, od.ProductID
FROM Orders o
JOIN [Order Details] od
  ON o.OrderID = od.OrderID
WHERE OrderDate = '7/4/1996'  -- Это - первая дата заказа, OrderDate, в системе
```

Выполнение этого запроса приводит к получению правильных результатов:

```
OrderDate                   ProductID
-------------------------   -----------
1996-07-04 00:00:00.000     11
1996-07-04 00:00:00.000     42
1996-07-04 00:00:00.000     72

(3 row(s) affected)
```

Но предположим для примера, что в данной системе регулярно удаляются устаревшие данные, но остается потребность в поиске ответа на тот же вопрос для составления автоматизированного отчета.

Итак, перед нами стоит задача автоматизации работы, поскольку нельзя предусмотреть выполнение запроса для определения того, за какую первую дату имеются данные в системе, после чего вручную включить полученную дату в следующий запрос. Тем не менее возможность автоматизации выполнения этой операции не исключена. Для этого достаточно применить следующий единственный оператор:

```
SELECT DISTINCT o.OrderDate, od.ProductID
FROM Orders o
JOIN [Order Details] od
  ON o.OrderID = od.OrderID
WHERE o.OrderDate = (SELECT MIN(OrderDate) FROM Orders)
```

Как оказалось, решение является простым и удобным. С помощью внутреннего запроса, `(SELECT MIN...)`, осуществляется выборка единственного значения, предназначенного для использования во внешнем запросе. Кроме того, поскольку в качестве операции сравнения задана операция сравнения на равенство, необходимо обеспечить, чтобы внутренний запрос возвращал значение только единственного поля одной строки, поскольку в противном случае возникает ошибка этапа прогона.

Запросы с вложенными подзапросами, которые возвращают несколько значений

По-видимому, наиболее широко применяются такие подзапросы, в которых в той или иной форме осуществляется выборка перечня допустимых значений, которые затем используются в запросе в качестве критериев.

Для ознакомления с примером такого применения вложенного подзапроса переключимся на использование базы данных `Pubs` (см. главу 5). Предположим, что требуется составить список всех магазинов, в которых покупателям предоставляются скидки. Напомним, что информация о магазинах хранится в таблице `Stores`, а сведения о скидках приведены в таблице `Discounts`.

Предположим, что вначале для решения этой задачи применяется следующий запрос:

```
USE PUBS
SELECT stor_id AS "Store ID", stor_name AS "Store Name"
FROM Stores
WHERE stor_id IN (SELECT stor_id FROM Discounts)
```

Как оказалось, выполнение этого запроса влечет за собой получение только одной строки, но любопытно то, что полученные результаты полностью совпадают с результатами выполнения внутреннего соединения в запросе, приведенном в главе 5:

```
Store ID       Store Name
----------     ---------------
8042           Bookbeat

(1 row(s) affected)
```

И действительно, запросы с вложенными подзапросами, подобные приведенному выше, почти всегда можно отнести к категории таких запросов, которые можно осуществить с использованием внутреннего соединения, а не вложенного оператора SELECT. Например, такие же результаты, как и в предыдущем запросе с вложенным подзапросом, можно получить после вызова на выполнение следующего простого оператора с конструкцией JOIN:

```
SELECT s.stor_id AS "Store ID", stor_name AS "Store Name"
FROM Stores s
JOIN Discounts d
  ON s.stor_id = d.stor_id
```

В целях обеспечения высокой производительности следует неизменно прибегать к использованию метода, основанного на соединении, если нет каких-либо весомых аргументов в пользу применения запроса с вложенным оператором SELECT. Дополнительная информация на эту тему приведена в конце настоящей главы.

Фактически в СУБД SQL Server предусмотрена возможность успешно справиться самостоятельно с решением этой задачи. В подавляющем большинстве ситуаций в СУБД SQL Server применительно к запросу с вложенным подзапросом действительно создается такой же план выполнения запроса, который использовался бы и для запроса с оператором соединения. Поэтому, учитывая данное обстоятельство, следует откровенно подчеркнуть, что чаще всего не наблюдается какое-либо различие в производительности при сравнении операторов с вложенными подзапросами и операторов с соединениями. Проблема заключается не в том, какие действия по выполнению запроса осуществляет СУБД, а в том, что уже неоднократно было сказано в этой книге. Если оказывается, что в СУБД для запросов того и другого типа используются разные планы выполнения запросов, то обычно обнаруживается, что вариант запроса с соединением обладает лучшими характеристиками, поэтому рекомендуется по умолчанию всегда использовать именно такие синтаксические конструкции.

Использование запроса с вложенным оператором SELECT для поиска висячих строк

Вложенный оператор SELECT такого типа, который рассматривается в данном разделе, почти идентичен приведенному в предыдущем примере, за исключением того, что в нем используется также операция NOT. Переходя к сравнению с синтаксической конструкцией соединения, можно отметить, что из-за этого различия применяемый оператор с вложенным подзапросом начинает в большей степени напоминать внешнее соединение, а не внутреннее.

Прежде чем перейти к изучению синтаксической конструкции искомого оператора с вложенным оператором SELECT, еще раз рассмотрим один из примеров внешнего соединения, приведенного в главе 5. В этом запросе предпринималась попытка выяснить, каким данным о магазинах из базы данных Pubs не соответствуют какие-либо строки с данными о скидках:

```
USE Pubs
SELECT s.Stor_Name AS "Store Name"
FROM Discounts d
RIGHT OUTER JOIN Stores s
           ON d.Stor_ID = s.Stor_ID
WHERE d.Stor_ID IS NULL
```

В результате были получены следующие пять строк:

```
Store Name
----------------------------------------
Eric the Read Books
Barnum's
News & Brews
Doc-U-Mat: Quality Laundry and Books
Fricative Bookshop

(5 row(s) affected)
```

Если речь идет о том, какие операторы должны, как правило, использоваться для решения указанной задачи, то основная рекомендация состоит в применении именно таких операторов. Но автор не может со всей уверенностью утверждать, что эта рекомендация действительно всегда применяется на практике. Вообще говоря, чтобы составить оператор с соединением, требуется предпринять более значительные умственные усилия, поэтому разработчики чаще всего вместо этого прибегают к использованию вложенных операторов SELECT.

Автор предлагает попытаться составить такой запрос с вложенным оператором SELECT самостоятельно, но должен предупредить читателя о том, что в ходе этого возникнет определенный нюанс. Закончив выполнение этой работы, ознакомьтесь с вариантом, предлагаемым автором.

По моему мнению, такой запрос должен выглядеть следующим образом:

```
SELECT stor_id AS "Store ID", stor_name AS "Store Name"
FROM Stores
WHERE stor_id NOT IN
   (SELECT stor_id FROM Discounts WHERE stor_id IS NOT NULL)
```

Выполнение этого запроса приводит к получению точно таких же пяти строк. Но я полагаю, что при первой попытке читатель не предусмотрел операцию сравнения IS NOT NULL во внутреннем запросе.

При решении подобной задачи необходимость включать или не включать операцию проверки IS NOT NULL зависит от того, допускается ли в рассматриваемом столбце применение NULL-значений, и от того, какие именно результаты требуется получить. Если в данном случае указанная операция сравнения не будет предусмотрена, то в конечном итоге сформируются неправильные результаты, на основании которых будет сделан ошибочный вывод, что нет таких магазинов, в которых не предоставляются скидки (тогда как в действительности такие магазины есть). Причина этого связана с тем, как осуществляется сравнение со NULL-значениями. Вообще говоря, если есть возможность появления NULL-значений в списке IN, то при подготовке запроса необходимо соблюдать предельную осмотрительность.

Связанные подзапросы

Прежде чем приступить к изложению темы настоящего раздела, автор хочет высказать настоятельную рекомендацию — обратите особое внимание на то, что здесь сказано! Обычно все разработчики обладают каким-то стандартным набором навыков, но тот, кто знает, как действуют связанные подзапросы и как они применяются, имеет в своем распоряжении гораздо более широкие возможности. Говоря о том, как важно изучить эту тему, автор стремится также подчеркнуть, что важно понять практическую значимость связанных подзапросов.

Связанные подзапросы относятся к категории таких инструментов, с помощью которых можно сделать невозможное возможным. Более того, с помощью связанных подзапросов часто можно превратить несколько строк кода в одну и добиться соответствующего повышения производительности. Но сложность освоения данной темы обусловлена тем, что вначале требуется овладеть принципиально иным стилем мышления по сравнению с тем, который обычно применяется в программировании. На первый взгляд кажется, что связанные подзапросы относятся к категории наиболее простых в изучении и освоении понятий языка SQL. Но после того как процесс изучения заканчивается, разработчик напрочь о них забывает, причем лишь потому, что сами понятия, лежащие в основе связанных подзапросов, противоречат складу его ума. Если же читатель относится к числу тех немногих счастливцев, которым удается закрепить в памяти принципы действия связанных подзапросов, то он, безусловно, становится также одним из немногих, которые способны решать задачи, которые кажутся не поддающимися решению. А когда придется бороться за каждую миллисекунду, чтобы повысить производительность запросов, то именно вы будете иметь в своем распоряжении гораздо более полный комплект инструментальных средств по сравнению с другими разработчиками.

Принципы работы связанных подзапросов

Как было описано выше, вложенные подзапросы выполняются как однонаправленные, а при выполнении связанных подзапросов, в отличие от вложенных подзапросов, информация движется в обоих направлениях, а не в одном. Внутренний запрос в операторе с вложенным подзапросом обрабатывается только один раз, после чего информация передается во внешний запрос, который также выполняется лишь одиножды. По существу, при этом с помощью внутреннего запроса вырабатывается единственное значение (или список значений), которое вполне можно было бы подставить во внешний запрос, введя его вручную.

С другой стороны, при выполнении связанных подзапросов внутренний запрос действует на основании информации, предоставленной внешним запросом, и наоборот. На первый взгляд такая организация работы может показаться не совсем понятной (снова возникает проблема "курицы" или "яйца"), но фактически осуществляется описанный ниже трехэтапный процесс.

- Внешний запрос получает строку, и эта строка передается во внутренний запрос.
- Внутренний запрос выполняется с учетом переданного в него значения (значений).
- Затем внутренний запрос передает во внешний запрос значения, сформированные из полученных в нем результатов, а во внешнем запросе эти значения используются для завершения намеченной в нем обработки.

Использование связанных подзапросов в конструкции WHERE

Автор признает, что приведенного выше описания может быть недостаточно, поэтому рассмотрим применение связанных подзапросов на примерах.

Вернемся к использованию базы данных Northwind и снова займемся составлением запроса, позволяющего узнать, какие заказы были введены в ту первую дату, за которую имеется информация о заказах, введенных в систему. Однако, предположим, что на этот раз необходимо учесть дополнительное требование – определить значения OrderID и OrderDate первого заказа, поступившего в систему от каждого заказчика. Иными словами, требуется определить, когда впервые каждый из заказчиков разместил в компании свой заказ и какие идентификаторы имеют эти заказы. Рассмотрим решение этой задачи последовательно.

Прежде всего для получения каждого набора искомых результатов необходимо определить значения OrderDate, OrderID и CustomerID. Всю эту информацию можно найти в таблице Orders, поэтому можно сделать вывод, что будущий запрос должен быть хотя бы отчасти основан на этой таблице.

Затем отметим, что для каждого заказчика необходимо определить, когда в системе впервые появились его заказы. Именно с этого момента начинаются сложности. Если бы применялся вложенный подзапрос, то с его помощью происходил бы поиск только самого первого значения даты во всей таблице, а теперь мы должны узнать, каковой была дата поступления первого заказа от каждого отдельного заказчика.

В этом не было бы ничего сложного, если бы допускалось использование двух отдельных запросов, поскольку достаточно было бы просто создать временную таблицу, а затем выполнить операцию соединения с этой таблицей.

Временная таблица полностью соответствует своему названию. Таковой является таблица, которая создается для временного использования, а после завершения соответствующего этапа обработки данных уничтожается. В этом отношении временные таблицы полностью аналогичны переменным. Описание временных таблиц выходит за рамки настоящей главы, но в следующих главах мы еще будем возвращаться к теме временных таблиц.

Решение, основанное на использовании временной таблицы, может выглядеть примерно таким образом:

```
USE Northwind
-- Получить список заказчиков и определить для каждого дату первого заказа
SELECT CustomerID, MIN((OrderDate)) AS OrderDate
INTO #MinOrderDates
FROM Orders
GROUP BY CustomerID
ORDER BY CustomerID

-- Выполнить необходимые дополнительные действия с этой информацией
SELECT o.CustomerID, o.OrderID, o.OrderDate
FROM Orders o
JOIN #MinOrderDates t
  ON o.CustomerID = t.CustomerID
  AND o.OrderDate = t.OrderDate
ORDER BY o.CustomerID

DROP TABLE #MinOrderDates
```

Выполнение этих операторов приводит к получению 89 строк, как показано ниже.

```
(89 row(s) affected)
```

```
CustomerID       OrderID            OrderDate
-------------    ---------------    -------------------------------
ALFKI            10643              1997-08-25 00:00:00.000
ANATR            10308              1996-09-18 00:00:00.000
ANTON            10365              1996-11-27 00:00:00.000
AROUT            10355              1996-11-15 00:00:00.000
BERGS            10278              1996-08-12 00:00:00.000
...
...
...
WHITC            10269              1996-07-31 00:00:00.000
WILMK            10615              1997-07-30 00:00:00.000
WOLZA            10374              1996-12-05 00:00:00.000

(89 row(s) affected)
```

Как уже было сказано, при выполнении того же сценарии на компьютере читателя могут быть получены результаты, отличные от приведенных в этом примере. Причина этого может быть связана с тем, что вы провели с данными базы данных `Northwind` *либо больше, либо меньше, экспериментов, чем автор.*

Итак, в данном случае формируются два полностью отдельных результирующих набора. На это указывает тот факт, что в результатах присутствуют две разные строки с указанием количества затронутых строк, `row(s) affected`. Но такая организация работы чаще всего отрицательно влияет на производительность. Этот вопрос будет рассматриваться более подробно в данной главе после описания других способов решения данной задачи.

Однако следует отметить, что иногда такой подход, основанный на использовании двух запросов, становится единственным способом добиться поставленной цели без применения курсора, но рассматриваемая задача не относится к такой категории.

Итак, было бы желательно объединить два приведенных выше запроса в один, но для этого необходимо найти способ поиска информации о каждом отдельном заказчике. Этой цели можно достичь, используя внутренний запрос, который выполняет поиск с учетом текущего значения `CustomerID`, полученного из внешнего запроса. После этого необходимо снова вернуть результат внутреннего запроса во внешний запрос, чтобы можно было выполнить выборку требуемых строк по данным о самой ранней дате заказа.

Оператор, созданный на основании изложенных требований, может выглядеть примерно так:

```
SELECT o1.CustomerID, o1.OrderID, o1.OrderDate
FROM Orders o1
WHERE o1.OrderDate = (SELECT Min(o2.OrderDate)
                      FROM Orders o2
                      WHERE o2.CustomerID = o1.CustomerID)
ORDER BY CustomerID
```

Применение этого оператора приводит к получению тех же 89 строк:

```
CustomerID       OrderID            OrderDate
-------------    -------------      -------------------------------
ALFKI            10643              1997-08-25 00:00:00.000
ANATR            10308              1996-09-18 00:00:00.000
```

```
ANTON              10365              1996-11-27 00:00:00.000
AROUT              10355              1996-11-15 00:00:00.000
BERGS              10278              1996-08-12 00:00:00.000
...
...
...
WHITC              10269              1996-07-31 00:00:00.000
WILMK              10615              1997-07-30 00:00:00.000
WOLZA              10374              1996-12-05 00:00:00.000
(89 row(s) affected)
```

В этом запросе заслуживают внимания несколько описанных ниже важных особенностей.

- В результатах обнаруживается только одно сообщение о количестве затронутых строк, `row(s) affected`. На этом основании можно с уверенностью сказать, что для выполнения был намечен только один план запроса.
- Внешний запрос (в данном примере) весьма напоминает по внешнему виду вложенный подзапрос, а внутренний запрос имеет явно определенную ссылку на внешний запрос (обратите внимание на использование псевдонима "o1").
- Псевдонимы применяются в обоих запросах (даже несмотря на то, что на первый взгляд во внешнем запросе псевдоним не требуется), поскольку псевдонимы необходимы для каждого явного обращения к столбцу из внешнего или внутреннего запроса (при этом либо внутренний запрос обращается к столбцу внешнего запроса, либо наоборот).

Обычно для начинающих разработчиков становится значительным источником путаницы последний пункт, касающийся необходимости использовать псевдонимы. Но истина заключается в том, что псевдонимы иногда действительно нужны, а иногда нет. Автор обычно не использовал псевдонимы во всех типах вложенных подзапросов, которые рассматривались в начальных разделах этой главы, а что касается связанных подзапросов, то в них неизменно применяются псевдонимы.

Существует надежное и простое правило — псевдоним необходимо предусматривать для любой таблицы (и относящихся к ней столбцов), на которую должна быть сформирована ссылка из внешнего запроса. Но проблема заключается в том, что при попытке составить запрос, руководствуясь этим правилом, может очень быстро возникнуть путаница в отношении того, для чего нужен псевдоним, а для чего нет. Поэтому самый оправданный подход состоит в том, чтобы предусматривать псевдонимы для всех таблиц. Это позволяет уверенно разбираться в том, из какой таблицы какого запроса поступает искомая информация.

В рассматриваемом примере сообщение `89 row(s) affected` появилось только один раз. Это связано с тем, что в запросе 89 строк действительно были затронуты только единожды. Даже на основании только этого наблюдения можно сделать вывод, что последняя версия с одним запросом должна, по-видимому, обладать более высоким быстродействием, чем версия с двумя запросами. Так оно и обстоит в действительности. Дополнительная информация на эту тему также будет приведена ниже.

В данном конкретном запросе внешний запрос ссылается на внутренний запрос в конструкции WHERE. Во внешнем запросе реализована также возможность запросить данные из внутреннего запроса для включения в список выборки.

При обычных обстоятельствах право использовать или не использовать псевдоним остается за разработчиком, а когда речь идет о связанных подзапросах, применение псевдонимов часто становится обязательным. Данный конкретный запрос представляет собой весьма наглядный пример, показывающий, чем обусловлена такая необходимость, поскольку в нем и внутренний, и внешний запросы основаны на одной и той же таблице. Очевидно, что оба запроса получают информацию друг от друга, поэтому без псевдонимов невозможно было бы указать, какой экземпляр данных таблицы представляет интерес в том и в другом случае.

Применение связанных подзапросов в списке выборки

Подзапросы могут также использоваться для получения в виде результатов выборки ответов немного другого рода. Необходимость в этом часто возникает, если искомая информация принципиально отличается по своему составу от остальной части данных, рассматриваемых в запросе (например, если необходимо выполнить агрегирование данных по одному столбцу, но желательно при этом избежать влияния выполнения операции агрегирования на результаты выборки данных из других столбцов).

В качестве примера подзапроса такого типа рассмотрим немного модифицированную версию запроса, который использовался в последнем разделе. Но в данном случае требуется определить только имена заказчиков и первую дату получения заказа от каждого из заказчиков.

Для составления требуемого запроса придется внести более значительные изменения, чем может показаться на первый взгляд. Прежде всего, теперь в результатах должно присутствовать имя заказчика, а это означает, что в запрос необходимо включить таблицу Customers. Кроме того, больше не требуется включать в запрос какое-либо условие, поскольку должны быть получены данные по всем заказчикам (без исключения), и требуется лишь узнать, какова дата получения первого заказа от этих заказчиков.

Фактически применение связанного подзапроса позволяет составить немного более простой запрос, чем тот, который использовался перед этим для решения указанной задачи. Новый вариант запроса выглядит следующим образом:

```
SELECT cu.CompanyName,
   (SELECT Min(OrderDate)
         FROM Orders o
         WHERE o.CustomerID = cu.CustomerID)
         AS "Order Date"
FROM Customers cu
```

Результаты выполнения этого запроса приведены ниже.

```
CompanyName                              Order Date
---------------------------------------  ------------------------------
Alfreds Futterkiste                      1997-08-25 00:00:00.000
Ana Trujillo Emparedados y helados       1996-09-18 00:00:00.000
Antonio Moreno Taqueria                  1996-11-27 00:00:00.000
Around the Horn                          1996-11-15 00:00:00.000
Berglunds snabbkop                       1996-08-12 00:00:00.000
Blauer See Delikatessen                  1997-04-09 00:00:00.000
...
...
...
```

```
White Clover Markets                    1996-07-31 00:00:00.000
Wilman Kala                             1997-07-30 00:00:00.000
Wolski Zajazd                           1996-12-05 00:00:00.000
(91 row(s) affected)
```

В приведенных выше сокращенных результатах это не показано, но просмотр всего объема полученных данных позволяет обнаружить большое количество строк, содержащих NULL-значения в столбце `Order Date`. С чем, по мнению читателя, это связано? Безусловно, причина появления NULL-значений состоит в том, что отсутствует строка в таблице `Orders`, соответствующая строке таблицы `Customers`, рассматриваемой как текущая (во внешнем запросе).

В связи с описанной ситуацией целесообразно немного отвлечься, чтобы рассмотреть чрезвычайно полезную функцию, позволяющую распознавать наличие NULL-значений, — `ISNULL()`.

Обработка данных, содержащих NULL-значения, с помощью функции ISNULL

В действительности количество функций, специально предназначенных для обработки данных, содержащих NULL-значения, невелико, но в рассматриваемой ситуации мы имеем возможность воспользоваться чрезвычайно полезной функцией такого типа — `ISNULL()`. Функция `ISNULL()` принимает в качестве параметра переменную (о чем речь пойдет в главе 11) или выражение, а затем проверяет, не имеет ли этот параметр неопределенное значение. Если параметр действительно имеет NULL-значение, то функция `ISNULL()` возвращает некоторое другое заранее заданное значение. Если же параметр функции имеет значение, отличное от `NULL`, функция возвращает именно это значение. Синтаксис вызова функции `ISNULL()` довольно простой:

```
ISNULL(<expression to test>, <replacement value if null>)
```

Примеры результатов, полученных при вызове функции `ISNULL()`, приведены в табл. 7.1.

Таблица 7.1. Примеры использования функции ISNULL()

Выражение ISNULL	Возвращаемое значение
`ISNULL(NULL, 5)`	5
`ISNULL(5, 15)`	5
`ISNULL(MyColumnName, 0) where MyColumnName IS NULL`	0
`ISNULL(MyColumnName, 0) where MyColumnName = 3`	3
`ISNULL(MyColumnName, 0) where MyColumnName ='Fred Farmer'`	Fred Farmer

Теперь рассмотрим, как можно применить эту функцию в данном примере запроса:

```
SELECT cu.CompanyName,
   ISNULL(CAST ((SELECT MIN(o.OrderDate)
         FROM Orders o
         WHERE o.CustomerID = cu.CustomerID)AS varchar), '  NEVER ORDERED')
         AS "Order Date"
FROM Customers cu
```

Перед этим неприемлемые результаты были получены в следующих двух строках:

```
...
FISSA Fabrica Inter. Salchichas S.A.      NULL
...
Paris specialites                         NULL
...
```

А после указанной доработки запроса формируются следующие, гораздо более полезные результаты:

```
...
FISSA Fabrica Inter. Salchichas S.A.      NEVER ORDERED
...
Paris specialites                         NEVER ORDERED
...
```

Обратите внимание на то, что для успешной реализации данного запроса пришлось применить также функцию `CAST()`. *Это потребовалось в связи с тем, что при выполнении рассматриваемого запроса возникают проблемы приведения типа и неявного преобразования. Поскольку при обработке первой строки происходит возврат действительной даты, принимается предположение, что столбец* `Order Date` *относится к типу* `DateTime`. *А после того как впервые происходит замена NULL-значения строковым значением* `NEVER ORDERED` *с помощью функции* `ISNULL`, *возникает ошибка, поскольку значение строки* `NEVER ORDERED` *невозможно преобразовать в значение с типом данных* `DateTime`. *Рекомендуем читателю учесть возможность применения функции* `CAST()`, *поскольку она позволяет справиться с небольшими сложностями, подобными описанной. Дополнительная информация о функции* `CAST()` *приведена ниже в данной главе.*

Таким образом, вы уже ознакомились со связанными подзапросами, которые предоставляют информацию и для конструкции `WHERE`, и для списка выборки. При желании, в одном и том же запросе могут применяться любые комбинации подзапросов того и другого типа.

Производные таблицы

Иногда в процессе обработки данных возникают такие ситуации, когда необходимо обеспечить работу с результатами запроса, но для этого требуется применить такой способ доступа к результатам, который фактически не сводится ни к одному из тех разновидностей подзапросов, которые были описаны до сих пор. В качестве примера можно указать такую ситуацию, что каждой строке в какой-то конкретной таблице может соответствовать несколько результатов, полученных с помощью подзапроса, и к этим результатам должна быть применена более сложная операция по сравнению с той, какая может быть предусмотрена в конструкции `IN`. По существу, к этому сводятся в основном такие ситуации, в которых требуется использовать в подзапросе операцию `JOIN`.

Именно при таких обстоятельствах приходится прибегать к применению конструкции SQL, менее широко распространенной по сравнению с другими, – **производной таблицы**. Производная таблица состоит из строк и столбцов результирующего набора некоторого запроса. (В конечном итоге результирующий набор обладает такими же свойствами, как и обычные таблицы, состоит из столбцов, строк, харак-

теризуется применением определенных типов данных, поэтому вполне допускает использование в качестве таблицы.)

Предположим, например, что требуется получить список заказчиков, заказавших определенный товар, скажем, шоколад, Chocolade. Решение этой задачи не представляет никакой сложности. Необходимый для этого запрос может выглядеть примерно таким образом:

```
SELECT c.CompanyName
 FROM Customers AS c
 JOIN Orders AS o
      ON c.CustomerID = o.CustomerID
 JOIN [Order Details] AS od
      ON o.OrderID = od.OrderID
 JOIN Products AS p
      ON od.ProductID = p.ProductID
 WHERE p.ProductName = 'Chocolade'
```

Итак, с первой задачей мы справились легко. А теперь рассмотрим более сложную задачу. Предположим, что требуется получить список всех заказчиков, которые заказали не только товар, обозначенный как Chocolade, но и вегетарианский паштет, Vegie-spread. Обратите внимание на то, что по условиям задачи требуется получить список заказчиков, заказавших оба этих товара. А в связи с решением этой задачи возникают сложности. Первой попыткой ее решения могло бы стать применение запроса с такой конструкцией:

```
WHERE p.ProductName = 'Chocolade' AND p.ProductName = 'Vegie-spread'
```

Но подобный запрос является абсолютно неприемлемым, так как каждая строка содержит сведения только об одном товаре, поэтому таких строк, в которых были бы одновременно указаны названия товаров Chocolade и Vegie-spread, в базе данных нет. Это означает, что попытка выборки подобных строк окончится неудачей (и действительно, после вызова запроса на выполнение формируется результирующий набор, не содержащий строк).

Фактически для решения данной задачи необходимо соединить результаты запроса по поиску покупателей товара Chocolade с результатами запроса по поиску покупателей товара Vegie-spread. А для соединения этих результатов, вполне очевидно, потребуется воспользоваться конструкциями, описанию которых посвящен настоящий раздел, — производными таблицами.

Для создания производной таблицы необходимо выполнить два описанных ниже условия.

- Заключить в круглые скобки запрос, который вырабатывает необходимый результирующий набор.
- Предусмотреть в запросе псевдоним для полученных результатов.

Итак, синтаксис оператора, позволяющего решить поставленную задачу, должен выглядеть примерно таким образом:

```
SELECT <select list>
FROM (<query that returns a regular resultset>) AS <alias name>
JOIN <some other base or derived table>
```

Воспользуемся этим замыслом для реализации поставленных требований, с учетом того, что необходимо также определить названия всех компаний, которые за-

казали и товар Chocolade, и товар Vegie-spread. Поэтому итоговый запрос должен выглядеть следующим образом:

```
SELECT DISTINCT c.CompanyName
FROM Customers AS c
JOIN
  (SELECT CustomerID
 FROM Orders o
 JOIN [Order Details] od
       ON o.OrderID = od.OrderID
 JOIN Products p
       ON od.ProductID = p.ProductID
 WHERE p.ProductName = 'Chocolade') AS spen
  ON c.CustomerID = spen.CustomerID
JOIN
  (SELECT CustomerID
 FROM Orders o
 JOIN [Order Details] od
       ON o.OrderID = od.OrderID
 JOIN Products p
       ON od.ProductID = p.ProductID
 WHERE ProductName = 'Vegie-spread') AS spap
  ON c.CustomerID = spap.CustomerID
```

Как оказалось, выполнение данного запроса приводит к получению сведений только об одном заказчике:

```
CompanyName
----------------------------------------
Ernst Handel
(1 row(s) affected)
```

Чтобы проверить правильность полученных результатов, достаточно просто выполнить запросы, относящиеся к двум производным таблицам отдельно, а затем сравнить полученные результаты.

Для данного конкретного запроса необходимо использовать ключевое слово DISTINCT. В противном случае не была бы исключена возможность получения многочисленных строк, относящихся к каждому заказчику. Например, компания Ernst Handel заказывала товар Vegie-spread дважды, поэтому было бы получено по одной строке для каждого заказа. А по условиям рассматриваемой задачи необходимо определить, какие заказчики заказали и тот, и другой товар, а не количество сделанных ими заказов.

Вполне очевидно, что с помощью производных таблиц удалось сформировать запрос, задача составления которого на первый взгляд казалась невозможной, и не только получить нужную информацию, но даже добиться вполне приемлемой производительности.

Но следует учитывать, что производные таблицы не позволяют решать все возможные задачи. Например, если результирующий набор становится довольно большим и количество строк, участвующих в операции соединения, достигает больших величин, то может потребоваться создать временные таблицы и сформировать на них индексы (производные таблицы не имеют индексов). Безусловно, возможность применения тех или других инструментальных средств зависит от ситуации, но, во всяком случае, производные таблицы позволят вам значительно расширить свой арсенал инструментальных средств.

Операция EXISTS

Автор рассматривает EXISTS как операцию, но в документации Books Online для обозначения EXISTS применяется лишь термин "ключевое слово". По-видимому, применение указанной трактовки обусловлено тем, что EXISTS не полностью соответствует определению понятия операции. Разумеется, EXISTS в такой же степени заслуживает названия "операция", как и операция, обозначаемая ключевым словом IN, но нельзя также отрицать, что обработка данных в ней происходит во многом иначе.

При использовании операции EXISTS возврат данных фактически не происходит. Вместо этого вырабатывается значение TRUE или FALSE, указывающее на то, существуют ли данные, которые соответствуют критериям, заданным в том запросе, на который распространяется действие операции EXISTS.

Перейдем непосредственно к рассмотрению примера, позволяющего ознакомиться с тем, как применяется эта операция. В данном случае речь идет о таком запросе, с помощью которого формируется список заказчиков, разместивших в компании по меньшей мере один заказ (общее количество размещенных заказов нас не интересует):

```
SELECT CustomerID, CompanyName
FROM Customers cu
WHERE EXISTS
   (SELECT OrderID
         FROM Orders o
         WHERE o.CustomerID = cu.CustomerID)
```

Выполнение этого запроса приводит к получению тех же 89 строк, которые были неоднократно получены в различных примерах этой главы:

```
CustomerID              CompanyName
----------------        ----------------------------------------
ALFKI                   Alfreds Futterkiste
ANATR                   Ana Trujillo Emparedados y helados
ANTON                   Antonio Moreno Taqueria
AROUT                   Around the Horn
BERGS                   Berglunds snabbkop
BLAUS                   Blauer See Delikatessen
...
...
...
WHITC                   White Clover Markets
WILMK                   Wilman Kala
WOLZA                   Wolski Zajazd
(89 row(s) affected)
```

Очевидно, что такую же задачу можно было бы легко решить с помощью соединения:

```
SELECT DISTINCT cu.CustomerID, cu.CompanyName
FROM Customers cu
JOIN Orders o
  ON cu.CustomerID = o.CustomerID
```

В частности, применение указанного синтаксиса, основанного на использовании операции соединения, привело бы к получению точно таких же результатов (не считая возможных различий, вызванных другим порядком сортировки). Поэтому напра-

шивается резонный вопрос, с чем связана необходимость в использовании этого нового синтаксиса. Ответ на этот вопрос весьма прост — с потребностями повышения производительности.

Если в запросе используется ключевое слово EXISTS, то СУБД SQL Server не приходится выполнять полное построчное соединение. Вместо этого СУБД просматривает строки до тех пор, пока не находит первое соответствие, после чего немедленно останавливается. Дело в том, что операция EXISTS возвращает истинное значение сразу после обнаружения первого соответствия, поэтому необходимость в выполнении дальнейшего просмотра отпадает.

Рассмотрим кратко ту же ситуацию под другим углом зрения. Предположим, что необходимо найти с помощью запроса информацию о тех заказчиках, которые не разместили в компании ни одного заказа. В соответствии с тем методом, который был основан на использовании соединения (см. главу 5), потребовалось бы внести весьма существенные поправки в тот подход, с помощью которого получается требуемый ответ. Прежде всего нужно было бы воспользоваться не внутренним соединением, а внешним, затем применить операцию сравнения, чтобы определить, содержат ли какие-либо строки с данными о заказах NULL-значения.

Таким образом, запрос, основанный на использовании соединения, выглядит следующим образом:

```
USE Northwind
SELECT c.CustomerID, CompanyName
FROM Customers c
LEFT OUTER JOIN Orders o
          ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL
```

Этот запрос возвращает две строки.

А чтобы добиться такого же изменения состава полученных результатов при использовании операции EXISTS, достаточно добавить только одно ключевое слово — NOT:

```
SELECT CustomerID, CompanyName
FROM Customers cu
WHERE NOT EXISTS
   (SELECT OrderID
         FROM Orders o
         WHERE o.CustomerID = cu.CustomerID)
```

Выполнение этого оператора приводит к получению тех же двух строк:

```
CustomerID              CompanyName
----------------        --------------------------------------------------
FISSA                   FISSA Fabrica Inter. Salchichas S.A.
PARIS                   Paris specialites

(2 row(s) affected)
```

Различие в производительности в данном случае становится еще более заметным по сравнению с внутренним соединением. Дело в том, что при выполнении оператора с ключевым словом NOT в СУБД SQL Server применяется операция отрицания к результатам той же операции EXISTS. И несмотря на то, что теперь используются не сами полученные результаты, а их отрицание, СУБД SQL Server по-прежнему

прекращает просмотр сразу же после обнаружения одной строки, соответствующей условию. Единственное различие состоит в том, что условием прекращения просмотра становится возврат из операции сравнения значения FALSE, а не TRUE. Во всем остальном, кроме производительности, запрос с конструкцией NOT EXISTS не отличается от запроса с внешним соединением.

Другие способы использования конструкции EXISTS

Одним из широко применяемых способов использования конструкции EXISTS является проверка существования таблицы перед вызовом на выполнение оператора ее создания. На основании результатов такой проверки может быть предусмотрено уничтожение существующей таблицы или внесение изменений в существующую таблицу с помощью оператора ALTER либо какого-то другого. Чаще всего применяется примерно такой способ осуществления указанного действия:

```
IF EXISTS (SELECT * FROM sysobjects WHERE id = object_
id(N'[dbo].[Shippers]') AND OBJECTPROPERTY(id, N'IsUserTable') = 1)
DROP TABLE [dbo].[Shippers]
GO
CREATE TABLE [dbo].[Shippers] (
 [ShipperID] [int] IDENTITY (1, 1) NOT NULL ,
 [CompanyName] [nvarchar] (40) NOT NULL ,
 [Phone] [nvarchar] (24) NULL
)
GO
```

Очевидно, что операция EXISTS не должна возвращать какого-либо иного значения, кроме TRUE или FALSE, а это означает, что данная операция может стать основой очень удобного условного выражения. В предыдущем примере было показано, что оператор DROP TABLE вызывается на выполнение, только если создаваемая таблица уже существует; в противном случае данная часть сценария исключается и происходит переход непосредственно к оператору CREATE. Это позволяет предотвратить возникновение в ходе выполнения сценария следующих двух ошибок. Во-первых, исключается возможность неудачного завершения попытки выполнения оператора CREATE в связи с тем, что создаваемый в нем объект уже существует (а такое неудачное завершение могло бы привести к возникновению других проблем, если операция создания таблицы входит в состав сценария, где должны быть созданы другие таблицы, зависящие от успешно созданной первой таблицы). Во-вторых, неудачно завершилась бы попытка уничтожения таблицы с помощью оператора DROP, поскольку эта таблица еще не существует (хотя такая ошибка не приводит к каким-либо неприятным последствиям, кроме выработки сообщения, которое может лишь смутить заказчика, устанавливающего у себя ваш программный продукт). Таким образом, применение операции EXISTS позволяет предотвратить возникновение обеих ошибок.

Итак, операция EXISTS может успешно применяться для автоматизации многих действия по сопровождению базы данных. В качестве еще одного примера применения указанной операции для данной цели рассмотрим сценарий CREATE, предназначенный для создания такого объекта, о котором часто забывают, разрабатывая средства автоматизации сопровождения, — самой базы данных. В документации этап создания базы данных часто отражается как часть не совсем понятных указаний, в которых содержатся примерно такие фразы, — “создайте базу данных 'xxxx'”.

Бессмысленность этих указаний подчеркивается тем, что пользователи (которые чаще всего не понимают, что делают) при их выполнении фактически указывают имя базы данных в одинарных кавычках, или создают недостаточно большую базу данных, или допускают целый ряд других возможных и весьма примитивных ошибок. Единственное, что может спасти пользователя в этой ситуации — это наличие отдела технической поддержки с квалифицированными сотрудниками.

Вместо этого автор предлагает предусматривать небольшой сценарий для создания таких объектов базы данных, которые могли бы войти в состав проекта Northwind. Но чтобы исключить возможность уничтожения образцовой базы данных Northwind, назовем создаваемую базу данных NorthwindCreate. Кроме того, отметим, что сам этот оператор создания содержит минимум конструкций, поскольку наша цель — проиллюстрировать применение операции EXISTS, а не команды CREATE:

```
USE master
GO
IF NOT EXISTS (SELECT 'True' FROM sys.databases WHERE name = 'NorthwindCreate')
BEGIN
   CREATE DATABASE NorthwindCreate
END
ELSE
BEGIN
   PRINT 'Database already exists. Skipping CREATE DATABASE Statement'
END
GO
```

Во время первого вызова на выполнение этого оператора на компьютере пользователя не должно быть базы данных NorthwindCreate (если только такая база данных не была создана по стечению обстоятельств до того, как мы приступили к описанию данной темы), поэтому будет получен примерно такой ответ:

```
Command(s) completed successfully.
```

Разумеется, эти результаты, формируемые системой, ничего не говорят о том, какие именно действия были выполнены, но по крайней мере можно сделать вывод, что выполнение используемого оператора завершилось успешно.

После этого вызовем тот же сценарий на выполнение во второй раз, после чего сразу станут заметными изменения, поскольку сообщение, сформированное сценарием, является гораздо более содержательным:

```
Database already exists. Skipping CREATE DATABASE Statement
```

Итак, в сценарий создания базы данных было внесено небольшое, незаметное на первый взгляд изменение, которое позволяет значительно упростить работу пользователей, занимающихся самостоятельно инсталляцией полученного ими программного продукта. Но таковыми могут стать не только конечные пользователи, которые приобрели ваш готовый программный продукт, но и вы сами. Так или иначе, оператор EXISTS позволяет предусмотреть в сценарии все возможные ситуации.

На основании изложенного можно сделать общий вывод, что операция EXISTS действительно является очень удобной. Она позволяет не только значительно повысить быстродействие некоторых запросов, но и упростить структуру определенных запросов и сценариев.

В качестве предостережения следует указать, что при использовании оператора EXISTS может быть легко допущена ошибка, обусловленная применением традиционного, процедурного подхода. Дело в том, что на первый взгляд может показаться, будто эта операция является допустимой конструкцией в запросах многих типов, т.е. исключает необходимость в использовании других программных средств (допустим, соединений), но и это правило имеет свои исключения. В частности, запрос, который применялся в качестве примера при описании производных таблиц, можно было бы также сформулировать с помощью ряда операций EXISTS (по одной для каждого товара), но, как оказалось, вариант запроса с производной таблицей в этом случае показывает вдвое более высокое быстродействие. Но еще раз отметим, что здесь имеет место исключение, а не правило, поскольку обычно запросы с операцией EXISTS превосходят по своей производительности запросы с производными таблицами. Следует лишь не забывать, что и это правило имеет исключения.

Совместное применение типов данных. Функции CAST и CONVERT

На практике функции CAST и CONVERT используются довольно часто. Безусловно. в данной главе об этих двух функциях уже было сказано несколько слов, но с учетом того, что они находят довольно широкое применение, мы должны более подробно описать их возможности.

Обе эти функции, CAST и CONVERT, обеспечивают преобразование типов данных. В определенном отношении обе они выполняют одинаковые действия, за исключением того, что функция CONVERT выполняет также некоторые преобразования, связанные с форматированием даты, которые не поддерживаются функцией CAST.

Из сказанного непосредственно следует, что функция CONVERT не только обеспечивает осуществление всех преобразований, предусмотренных функцией CAST, но и выполняет преобразование даты. В связи с этим возникает вопрос – чем обусловлена необходимость использования функции CAST? Ответ на этот вопрос является простым – функция CAST совместима со стандартом ANSI, а функция CONVERT не предусмотрена стандартом ANSI. В этом и состоит их различие.

Ниже показан синтаксис вызова обеих этих функций.

```
CAST (expression AS data_type)
CONVERT(data_type, expression[, style])
```

Очевидно, что обе эти функции по существу имеют одинаковый синтаксис, если не считать того, что в них предусмотрена разная последовательность параметров, а при вызове функции CONVERT предусмотрена возможность задать опцию форматирования (с помощью параметра style).

Функции CAST и CONVERT позволяют осуществлять широкий перечень преобразований типов данных, которые могут потребоваться, если в СУБД SQL Server необходимые преобразования не осуществляются неявно. Например, очень часто возникает необходимость преобразовать числовые данные в строковые. Рассмотрим следующий пример:

```
SELECT 'The Customer has an Order numbered ' + OrderID
FROM Orders
WHERE CustomerID = 'ALFKI'
```

Выполнение этого оператора приводит к возникновению такой ошибки:

```
Msg 245, Level 16, State 1, Line 1
Conversion failed when converting the varchar value 'The Customer has an
Order numbered ' to data type int.
```

Внесем корректировку в этот код, чтобы в нем вначале выполнялось преобразование числовых данных:

```
SELECT 'The Customer has an Order numbered ' + CAST(OrderID AS varchar)
FROM Orders
WHERE CustomerID = 'ALFKI'
```

После этого будет получен требуемый результат:

```
------------------------------------------------------------
The Customer has an Order numbered 10643
The Customer has an Order numbered 10692
The Customer has an Order numbered 10702
The Customer has an Order numbered 10835
The Customer has an Order numbered 10952
The Customer has an Order numbered 11011

(6 row(s) affected)
```

Но результаты преобразования не всегда становятся такими наглядными. Предположим, например, что необходимо преобразовать значение столбца с временной отметкой в обычное число. Временная отметка — это просто двоичное число, поэтому в действительности задача преобразования не является такой уж сложной:

```
CREATE TABLE ConvertTest
(
  ColID   int   IDENTITY,
  ColTS   timestamp
)
INSERT INTO ConvertTest
  DEFAULT VALUES

SELECT ColTS AS "Uncoverted", CAST(ColTS AS int) AS "Converted" FROM
ConvertTest
```

Выполнение этого оператора приводит к получению примерно таких данных (в зависимости от обстоятельств могут быть получены другие значения):

```
(1 row(s) affected)
Uncoverted              Converted
-----------------       -------------
0x00000000000000C9      201
(1 row(s) affected)
```

Кроме того, с помощью функции CAST можно обеспечить преобразование дат:

```
SELECT OrderDate, CAST(OrderDate AS varchar) AS "Converted"
FROM Orders
WHERE OrderID = 11050
```

Выполнение этого оператора приводит к получению результатов, аналогичных приведенным ниже (конкретно применяемое форматирование может зависеть от опций конфигурации определения системной даты).

```
OrderDate                  Converted
-------------------------  ----------------------------
1998-04-27 00:00:00.000    Apr 27 1998 12:00AM
(1 row(s) affected)
```

В данном случае заслуживает внимания то, что преобразование даты выполняется так, как предусмотрено в функции CAST; такой контроль над форматированием, который предоставляет функция CONVERT, отсутствует. А пример применения последней функции приведен ниже.

```
SELECT OrderDate, CONVERT(varchar(12), OrderDate, 111) AS "Converted"
FROM Orders
WHERE OrderID = 11050
```

Выполнение этого оператора приводит к получению следующих результатов:

```
OrderDate                  Converted
-------------------------  ---------------
1998-04-27 00:00:00.000    1998/04/27

(1 row(s) affected)
```

Очевидно, что эти результаты весьма отличаются от тех, которые получены с помощью функции CAST. А в действительности мы могли бы преобразовать эту дату в любой из 34 форматов с двух- или четырехзначным обозначением года.

```
SELECT OrderDate, CONVERT(varchar(12), OrderDate, 5) AS "Converted"
FROM Orders
WHERE OrderID = 11050
```

Это приводит к получению таких результатов:

```
OrderDate                  Converted
-------------------------  -------------
1998-04-27 00:00:00.000    27-04-98

(1 row(s) affected)
```

От пользователя требуется лишь указать код в качестве последнего параметра функции CONVERT (в предыдущем примере был задан код 111, соответствующий японскому стандарту, с четырехзначным обозначением года, а в последнем примере – значение 5, соответствующее итальянскому стандарту, с двухзначным обозначением года). Этот последний параметр указывает требуемый формат. Все коды, превышающие 100, обусловливают применение четырехзначного обозначения года, а все коды меньше 100, за несколькими исключениями, соответствуют двухзначному обозначению года. Перечень всех доступных форматов можно найти в документации *Books Online* под заголовком *CONVERT or CASE*.

Следует учитывать, что для решения печально знаменитой проблемы двухтысячного года было внесено несколько изменений. Одно из этих изменений состояло в том, что можно задавать точку разделения, которая будет использоваться в СУБД SQL Server для определения того, следует ли добавлять спереди к двухзначному обозначению значение 20 или 19.

По умолчанию используется точка разбиения 49/50. В связи с этим двухзначное обозначение года, равное 49 или меньше, будет преобразовываться путем добавления цифр 20 спереди. К любому другому обозначению года, превышающему это число, будут добавляться цифры 19. Указанное значение точки разбиения можно изменить путем корректировки конфигурации сервера базы данных.

Вопросы повышения производительности

Выше уже были приведены некоторые общие рекомендации по повышению производительности, но, как и в отношении большинства практических вопросов, этого недостаточно, чтобы добиться реального успеха. Поэтому в настоящем разделе будет приведен своего рода краткий справочник по проблемам повышения производительности запросов. Приведенные здесь указания помогут выбрать именно такие типы запросов, которые в наибольшей степени подходят для той или другой ситуации.

Мы снова сталкиваемся с одной из проблем, при рассмотрении которой необходимо подвергнуть критическому анализу сложившееся положение дел. На сей раз объектом становится непродуманное использование "универсальных" правил.

В настоящем разделе речь идет о том, как обычно складывается процесс обработки данных. В данном контексте слово "обычно" становится чрезвычайно выразительным. В языке SQL, как и во многом другом, лишь немногие правила можно считать безусловно истинными. Но когда дело касается повышения производительности SQL Server, необходимо выбирать только такие средства языка SQL, которые надежно обеспечивают достижение поставленной цели.

Иначе говоря, многое зависит от того, насколько важно добиться высокой производительности данного конкретного запроса. Если производительность действительно является решающим фактором, то не ограничивайтесь использованием лишь общепринятых правил. Руководствуйтесь ими как отправной точкой, а затем снова и снова экспериментируйте и отыскивайте все более эффективные подходы.

Сравнение возможностей подзапросов и соединений

Выше в настоящей главе уже было сказано, что автор однажды участвовал в дискуссии со своим сотрудником по поводу того, какие конструкции, подзапросы или соединения являются более приемлемыми. Взгляды, высказанные обоими участниками обсуждения этого вопроса, оказались прямо противоположными. Как и следовало ожидать в таких обстоятельствах, мы оба оказались в определенной степени правы (а из этого следует, что и в какой-то степени неправы).

Давно сложившаяся, традиционная точка зрения на подзапросы всегда состояла в том, что вместо них при любой возможности гораздо лучше использовать соединения. При определенных обстоятельствах такое мнение является абсолютно правильным, а в других условиях многое зависит от широкого набора факторов. В табл. 7.2 приведены сведения о том, в каких ситуациях применяются подзапросы и соединения, и даны рекомендации по выбору наиболее рационального способа организации работы.

Таблица 7.2. Рекомендации по выбору наиболее рационального способа организации работы в различных ситуациях

Ситуация	Рекомендуемый способ организации работы
Предполагается, что значение, возвращаемое подзапросом, должно быть одинаковым для каждой строки во внешнем запросе	Рекомендуется предварительное выполнение запроса. Объявить переменную, а затем осуществить выборку необходимого значения в эту переменную. Это позволяет выполнить предполагаемый подзапрос только единожды, а не столько раз, сколько имеется строк во внешней таблице
Обе таблицы являются относительно небольшими (скажем, 10 000 строк или меньше)	Рекомендуется применение подзапросов. Точные причины мне неизвестны, но я сам неоднократно проверял эту рекомендацию, и в каждом случае она в значительной степени оправдывалась. По-видимому, поиск требует меньших издержек по сравнению с соединением
После проверки всех критериев в операции сравнения должно быть возвращено только одно значение	Рекомендуется применение подзапросов. И в этом случае гораздо меньше издержек требует поиск только одной строки и подстановка соответствующего ей значения, чем выполнение соединения по всей таблице
После проверки всех критериев в операции сравнения должно быть возвращено относительно немного значений; на столбце поиска не задан индекс	Рекомендуется применение подзапросов. Поиск по одному или даже по нескольким столбцам обычно требует меньших затрат ресурсов, чем хешированное соединение
Поисковая таблица является относительно небольшой, но базовая таблица велика	По возможности, следует применять вложенные подзапросы; в ином случае соединения являются более предпочтительными, чем связанные подзапросы. При использовании подзапросов поиск осуществляется только единожды, а издержки относительно малы. С другой стороны, при использовании связанных подзапросов поиск многократно повторяется в цикле. При указанных обстоятельствах наилучшим вариантом чаще всего становится соединение
Выбор между связанным подзапросом и соединением	Рекомендуется применение соединения. По существу, при выполнении связанного подзапроса создается вложенный цикл. Это может повлечь за собой непроизводительное расходование весьма значительных ресурсов. В большинстве ситуаций соединения обеспечивают гораздо более высокое быстродействие по сравнению с курсорами, но все равно остаются менее производительными по сравнению с другими возможными способами обработки данных
Выбор между производными таблицами и другими программными средствами	Применение производных таблиц обычно влечет за собой значительное повышение издержек, поэтому, предпринимая попытку организовать с их помощью обработку данных, необходимо действовать осмотрительно. Следует помнить, что создание (или, так сказать, "производство") подобных таблиц осуществляется лишь единожды, после чего производные таблицы хранятся в памяти. Это означает, что основная часть издержек обусловлена первоначальными затратами на создание таблиц и отсутствием индексов даже на очень крупных результирующих наборах. Вообще говоря, применение производных таблиц может способствовать либо понижению, либо повышению производительности; это зависит от конкретных условий. Таким образом, прежде чем приступить к созданию основанного на них программного обеспечения, необходимо тщательно обдумать все обстоятельства
Выбор между операторами `EXISTS` и другими программными средствами	Рекомендуется применение операторов `EXISTS`. Операторы `EXISTS` позволяют прекратить поиск после обнаружения первого же значения, соответствующего критерию поиска, и в случае необходимости перейти к выполнению следующей операции поиска. Это способствует значительному сокращению издержек

В табл. 7.2 приведены лишь самые основные рекомендации. Количество возможных ситуаций и соответствующих им способов организации работы является практически бесконечным.

> Невозможно переоценить важность следующей рекомендации: при малейшем сомнении предусматривать достаточно полную проверку и сопоставление всех возможных решений предстоящей задачи, особенно если производительность является решающим фактором. Разумеется, чаще всего "универсальные" правила вполне оправдываются, но так бывает не всегда. Проверка выбранного решения позволяет убедиться в его правильности.

Резюме

По-видимому, способы организации запросов, описанные в главах 3 и 4, охватывают 80% или даже больше ситуаций, связанных с применением запросов, с которыми приходится сталкиваться на практике, но остальные 20% могут оказаться буквально не поддающимися решению с использованием широко известных приемов программирования. Иногда возникают настолько сложные задачи, что нелегко даже понять, как приступить к формированию запроса, который позволит найти требуемый ответ, а в других случаях никак не удается добиться приемлемой производительности конкретного запроса или хранимой процедуры. Так или иначе, на практике не всегда удается достичь поставленной цели с помощью простых запросов и соединений. Требуется нечто большее, и мы надеемся, что средства, описанные в настоящей главе, позволят вам приобрести небольшой дополнительный арсенал, позволяющий справляться с такими трудными ситуациями.

Упражнения

7.1. Напишите запрос, который возвращает сведения о том, какова дата приема на работу каждого из служащих компании `Northwind`, в формате `MM/DD/YY`.

7.2. Напишите отдельные запросы с использованием соединения, подзапроса, а затем оператора `EXISTS`, чтобы составить список всех заказчиков компании `Northwind`, которые не разместили в ней свои заказы.

7.3. Покажите пять самых последних заказов, сделанных заказчиком, который потратил больше 25 000 долларов на приобретение товаров в компании `Northwind`.

8

Нормализация и другие важные проблемы проектирования

В предыдущих главах были приведены многочисленные примеры таблиц, а в данной главе речь пойдет о том, в каких целях в конечном итоге создаются таблицы. За редкими исключениями в настоящей книге в основном рассматриваются таблицы, предназначенные для оперативной обработки транзакций (OnLine Transaction Processing – OLTP). Безусловно, определенное внимание было уделено описанию различий между оперативной обработкой транзакций и оперативной аналитической обработкой (OnLine Analytical Processing – OLAP), которая представляет собой еще одно важное направление применения баз данных, но данная книга главным образом посвящена описанию проектов таблиц, в наибольшей степени предназначенных для самого основного направления использования баз данных – OLTP. В связи с этим рассматриваемые примеры таблиц базы данных чаще всего имели такую организацию, которую принято называть третьей нормальной формой.

В настоящей главе будет приведено достаточно подробное описание принципов нормализации. А на данный момент достаточно отметить, что нормализацией называется способ преобразования структуры данных в логически обоснованную форму, не содержащую повторяющихся данных, которая позволяет легко восстановить исходную структуру данных. Иными словами, данные в результате проведения нормализации принимают определенную стандартную структуру. Кроме вопросов нормализации, в этой главе рассматриваются основные характеристики баз данных OLTP и OLAP. Следует отметить, что в нормализованной базе данных ограничения имеют свою специфику, поэтому в завершение главы будет приведен ряд примеров применения таких ограничений.

По-видимому, настоящая глава является наиболее сложной для понимания по сравнению со всеми другими главами книги, поскольку при описании представленных в ней тем приходится сталкиваться с противоречием. Дело в том, что некоторые концепции, используемые в этой главе, относятся к тематике, которая будет рассматриваться позже, такой как триггеры и хранимые процедуры. С другой стороны, к изучению указанных тем сложно перейти, не достигнув понимания того, какую роль они играют в проектировании базы данных.

Автор настоятельно рекомендует после прочтения данной главы еще раз вернуться к ней позже, усвоив материал нескольких последующих глав.

Таблицы

Прежде всего необходимо рассмотреть формальное определение понятия таблицы, поскольку таблица – это наиболее важный объект любой базы данных.

Таблица – это коллекция экземпляров данных, имеющих несколько общих **атрибутов** и сгруппированных по **строкам** и **столбцам**. Еще одна трактовка понятия таблицы состоит в том, что таблица представляет собой совокупность практически значимых данных (такие объекты данных именуются **сущностями**), которые объединены **связями** с информацией, находящейся в других таблицах. Изображение различных сущностей (экземпляров данных) и связей (родительско-дочерних зависимостей между экземплярами данных) обычно называют **диаграммами "сущность–связь"** (или **ER-диаграммами**). Иногда вместо термина "ER-диаграмма" применяется сокращенное обозначение ERD (ER Diagram).

Связи позволяют переходить от одной таблицы к другой и формировать новые (обычно временные) таблицы из двух или нескольких таблиц, соединенных связями (некоторые сведения по этой теме уже были приведены в главах 4 и 5). А вся коллекция взаимосвязанных сущностей рассматривается как база данных.

Нормализация данных

Нормализация представляет собой основу проектирования современных баз данных OLTP. Понятие нормализации было сформулировано на основе определения реляционной базы данных. Оба эти понятия были впервые предложены в 1969 году в статье Э.Ф. Кодда (E.F. Codd), сотрудника компании IBM. Кодд сформулировал определение базы данных как объекта, "состоящего из совокупности неупорядоченных таблиц, к которым могут применяться непроцедурные операции, возвращающие таблицы".

Из этого определения базы данных следует несколько важных выводов.

- Порядок строк в таблицах не имеет значения.
- Должна быть предусмотрена возможность применять к таблицам реляционные операции (в своей работе Кодд называл таблицы "отношениями" – relation).
- Применение реляционных операций к таблицам должно обеспечивать возможность создания виртуальных таблиц, соответствующих предъявленным требованиям.

Дальнейшее развитие предложенного подхода к рассмотрению базы данных как состоящей из отношений привело к становлению понятия нормализации.

Нормализация относится к числу наиболее часто упоминаемых и вместе с тем неправильно трактуемых понятий в программировании. Каждый разработчик программного обеспечения баз данных считает себя полностью освоившим принципы нормализации, притом что действительно многие разбираются в этом вопросе, по крайней мере теоретически. Но, к сожалению, часто на этом так все и заканчивается. Изучение основных методов нормализации становится для некоторых проектировщиков лишь показателем того, что они "действительно" могут претендовать на звание архитектора базы данных. Однако подобные специалисты только упоминают нормальные формы в своих устных заявлениях, но не используют на практике. На самом деле нужно не рассуждать о необходимости нормализации, а применять ее как один из этапов создания проекта базы данных. Иногда необходимо провести процесс нормализации данных до последней возможной степени, а в других условиях более оправданным становится решение преднамеренно денормализовать часть данных. К тому же в ходе осуществления самого процесса нормализации часто можно достичь намеченной цели создания нормализованной базы данных несколькими разными способами.

По мнению автора, нормализацию следует рассматривать как как один из теоретически обоснованных подходов к усовершенствованию структуры базы данных. Решение о том, следует ли применять тот или иной метод нормализации, должно приниматься с учетом возможности создания на основе принципов нормализации такого проекта базы данных, который окажется полностью приемлемым с точки зрения практики. Не нужно слепо верить приведенным в книгах (в то числе и в этой) категорическим рекомендациям, касающимся того, что следует делать в той или другой ситуации; поступайте так, как диктует ситуация, в которой вы находитесь. Книга позволяет автору лишь донести свои знания до читателя, но не дает возможности показать, как эти знания должны применяться в действительности (во всяком случае, это невозможно показать на бумаге). Вы должны сами достичь глубокого понимания основных понятий нормализации и на этой основе неуклонно добиваться намеченной цели создания наиболее подходящего проекта базы данных.

Перейдем к описанию нормальных форм. Прежде всего следует отметить, что количество теоретически обоснованных нормальных форм равно шести. Однако многие рассматривают в качестве полностью нормализованной такую базу данных, нормализация которой проведена вплоть до третьей нормальной формы. Указанный подход является вполне оправданным, поскольку нормальные формы выше третьей в действительности применяются довольно редко. Поэтому в настоящей книге главным образом рассматриваются первые три нормальные формы. Тем не менее определенное внимание в данной главе будет также уделено трем остальным нормальным формам.

В предыдущих главах рассматривались способы создания первичных ключей и были приведены некоторые соображения в пользу их применения в таблицах. В частности, первичные ключи позволяют однозначно идентифицировать каждую строку, благодаря чему обеспечивается возможность точно задавать строки, на которые распространяется действие определенной операции. Понятие нормализации полностью основано на определении самого первичного ключа и столбцов, входящих в состав первичного ключа. Поэтому применительно к нормализации часто приходится слышать следующую краткую формулировку:

Основой нормализации являются ключи, только ключи и ничего кроме ключей.

Иногда встречается также забавное дополнение к этой формуле:

Основой нормализации являются ключи, только ключи и ничего кроме ключей, да поможет мне Кодд![1]

В этом и заключается предельно краткая сводка рекомендаций по проведению нормализации до третьей нормальной формы. После того как обнаруживается, что все столбцы таблицы зависят исключительно от первичного ключа (рассматриваемого как единое целое, независимо от количества входящих в него столбцов), можно сделать вывод о достижении третьей нормальной формы.

В следующих разделах рассматриваются различные нормальные формы и описано назначение каждой из них.

Предварительные сведения

Еще до того, как будет предпринята попытка преобразовать данные в первую нормальную форму, должны быть выполнены определенные условия. Мало того, каждая таблица, подлежащая преобразованию в первую нормальную форму, должна соответствовать перечисленным ниже требованиям, поскольку в противном случае ее нельзя будет рассматривать как компонент реляционной базы данных, в полной степени соответствующий определению сущности.

- Таблица должна представлять одну и только одну сущность (не допускается никаких попыток удаления части атрибутов одной сущности и добавления атрибутов другой сущности).
- Все строки должны быть уникальными, и на таблице должен быть задан первичный ключ.
- Порядок расположения столбцов и строк не должен иметь значения.

Таким образом, необходимо прежде всего правильно определить сущности. Применительно к некоторым из них решение указанной задачи становится довольно несложным, а другие сущности требуют для своего распознавания гораздо больше усилий. Чаще всего в процессе нормализации определения многих сущностей становится более очевидными и точными. По крайней мере, необходимо прежде всего выявить сущности, поддающиеся идентификации.

Проводя аналогию с объектно-ориентированным программированием, можно отметить, что большинство логических сущностей верхнего уровня близки по своему назначению объектам в объектной модели.

Еще раз рассмотрим чрезвычайно простую модель, о которой уже шла речь в данной книге, — модель системы сбыта. Прежде всего следует отметить, что в рассматриваемом примере нас не интересуют все возможные нюансы организации данных и не играет даже никакой роли вопрос о том, какие столбцы должны быть предусмотрены в таблице. Важно лишь определить, как должны быть выделены основные сущности, применяемые в системе.

На первом этапе необходимо определить, что лежит в основе процесса обработки данных. А в конечном итоге должно быть создано по одной сущности в расчете на каждую элементарную единицу представления данных, применимую для представления требуемых данных в процессе обработки. Применительно к системе сбыта про-

[1] Игра слов — Codd и God. — *Примеч. пер.*

цесс обработки данных начинается с того, что заказчик звонит или приходит лично, после чего обращается к служащему, который принимает заказ (а после оплаты заказа выписывает счет-фактуру).

Таким образом, при проведении первой итерации процедуры нормализации может быть выявлена одна сущность — заказ (обозначим сущность, представляющую заказы, как `Orders`).

По мере приобретения опыта в проведении процесса нормализации разработчику все чаще удается с самого начала выявлять все больше и больше сущностей, даже при первой итерации. Но рассматриваемый пример предназначен для того, чтобы показать, как другие необходимые сущности обнаруживаются в процессе нормализации.

После определения сущностей необходимо приступить к подготовке исходного варианта набора столбцов, а затем на основании анализа перечня столбцов задать первичный ключ. Напомним, что первичный ключ предоставляет уникальный идентификатор для каждой строки.

Этап выявления первичного ключа начинается с просмотра списка столбцов и определения **потенциальных ключей**. В список потенциальных ключей должны войти все столбцы, которые могут в принципе использоваться для однозначной идентификации каждой строки, относящейся к рассматриваемой сущности. После этого для использования в качестве первичного ключа назначается какой-то отдельно взятый ключ из списка потенциальных ключей. Другого простого и универсального способа принятия решения о том, какой столбец (столбцы) должен стать первичным ключом, не предусмотрено (в этом заключается одна из многих причин, под влиянием которых обнаруживается такое широкое разнообразие проектов баз данных, предназначенных для хранения одной и той же основной информации, если эти проекты подготовлены разными разработчиками). В некоторых случаях не удается найти даже один-единственный потенциальный ключ, поэтому возникает необходимость ввести дополнительный столбец, предназначенный для решения этой задачи (напомним, что для этой цели применяются столбцы типа `identity` и столбцы, данные в которые вводятся с помощью функции `rowguid()`).

Пример создания таблицы `Orders` уже рассматривался в предыдущей главе, но в настоящей главе для иллюстрации процесса нормализации вначале рассмотрим наиболее общую реализацию таблицы `Orders` в виде такого устаревшего средства представления данных, как плоский файл (табл. 8.1).

Таблица 8.1. Вариант представления данных, содержащихся в заказах, в виде плоского файла

Orders
OrderNo
CustomerNo
CustomerName
CustomerAddress
CustomerCity
CustomerState
CustomerZip
OrderDate
ItemsOrdered
Total

Безусловно, в этом плоском файле может быть представлено все содержимое таблицы `Orders`, кроме того, с точки зрения практики, каждый заказ должен иметь уникальный порядковый номер, поэтому вполне допустимо использовать столбец `OrderNo` в качестве столбца первичного ключа.

Итак, задача определения исходной сущности решена. Применительно к данной сущности не выдвигаются какие-либо требования, касающиеся упорядочения столбцов (в соответствии с общепринятым соглашением, таблицы обычно изображаются в таком виде, что столбец (столбцы) первичного ключа представлен в первую очередь, но, строго говоря, применяемый способ изображения таблицы не должен рассматриваться как способ определения порядка расположения столбцов). Кроме того, в самой компоновке таблицы, приведенной в табл. 8.2, отсутствуют какие-либо признаки, указывающие на то, каким должно быть упорядочение строк. Поэтому данная таблица (по крайней мере, на первый взгляд) представляет только одну сущность. Иными словами, мы можем заняться осуществлением процесса нормализации (а фактически мы уже в определенном смысле приступили к решению этой задачи).

Первая нормальная форма

Первая нормальная форма (First Normal Form — **1NF**) предназначена для устранения повторяющихся групп данных и обеспечения **неразрывности данных** (обеспечения того, чтобы данные не содержали ссылки на другие данные и были независимыми). Наиболее целесообразный способ достижения этой цели состоит в осуществлении следующих этапов: создать первичный ключ (данный этап уже выполнен), после чего перемещать все повторяющиеся группы данных в новые таблицы, создавать для этих таблиц новые ключи, и т.д. Кроме того, необходимо разделить на два или несколько столбцов все столбцы, характеризующиеся тем, что в них присутствуют многозначные данные, которые в различных строках представлены в разных комбинациях.

В приложениях на основе плоских файлов, которые в прошлом находили самое широкое распространение, повторяющиеся данные встречались очень часто (причем не реже, чем многочисленные фрагменты информации в одном столбце). Однако в связи с этим возникали сложные проблемы, описанные ниже.

- В то время дисковая память стоила чрезвычайно дорого, а из-за того, что одинаковые данные повторно записывались на внешнее устройство по нескольку раз, значительный объем пространства на диске затрачивался впустую. В последнее время стоимость дисковой памяти существенно снизилась, поэтому указанная проблема перестала быть столь острой.
- Чем больше повторяющихся данных, тем больше объем перемещаемых данных и больше количество операций ввода-вывода. Это означает, что в связи с необходимостью передавать по шине данных и (или) по сети крупные блоки данных снижается производительность. Указанный фактор может оказывать весьма заметное отрицательное влияние на производительность, даже несмотря на то, что современные технические средства обладают гораздо более высоким быстродействием.
- Если в разных строках с данными встречаются повторяющиеся данные, то такие данные часто не согласуются, по той причине, что когда-то была нарушена синхронизация при обновлении данных, а это приводит к появлению расхождений в данных и в общем вызывает нарушение целостности данных.

- При выборке данных из столбца, содержащего многозначные значения, приходится вначале осуществлять процедуру извлечения одного значения из нескольких (а это происходит чрезвычайно медленно).

На данный момент в таблице, структура которой показана в табл. 8.1, имеется много столбцов, причем вполне можно было бы добавить к ней еще несколько столбцов. Тем не менее применение такой таблицы предоставляет определенные удобства, поскольку появляется возможность получить в одном месте всю требуемую информацию о заказах.

Однако так кажется только на первый взгляд. Чтобы узнать, как фактически должна быть организована работа при использовании подобной таблицы, рассмотрим примеры того, как могут выглядеть данные в таблице заказов (табл. 8.2). Следует отметить, что при подготовке настоящей книги к печати пришлось удалить несколько столбцов, для того чтобы вся информация поместилась по ширине на одной странице, но от этого суть дела не меняется.

Таблица 8.2. Таблица с данными заказов

OrderNo	OrderDate	CustomerNo	CustomerName	CustomerAddress	ItemsOrdered
100	1/1/99	54545	ACME Co	1234 1st St.	5-1A4536, Flange, 7lbs, $75; 4-OR2400, Injector, .5lbs, $108; 4-OR2403, Injector, .5lbs, $116; 1-4I5436, Head, 63lbs, $750
101	1/1/99	12000	Sneed Corp.	555 Main Ave.	1-3X9567, Pump, 5lbs, $62.50
102	1/1/99	66651	ZZZ & Co.	4242 SW 2nd	7-8G9200; Fan, 3lbs, $84; 1-8G5437, Fan, 3lbs, $15; 1-3H6250, Control, 5lbs, $32
103	1/2/99	54545	ACME Co	1234 1st St.	40-8G9200, Fan, 3lbs, $480; 1-2P5523, Housing, 1lbs, $165; 1-3X9567, Pump, 5lbs, $42

Для обеспечения нормализации данной таблицы необходимо решить целый ряд проблем. Безусловно, для этой таблицы предусмотрен вполне применимый первичный ключ (и действительно, при формировании этой таблицы учтен значительный опыт, ведь ключи использовались задолго до появления реляционных систем), но, как указано ниже, два других основных требования к таблице, находящейся в первой нормальной форме, не выполнены.

- В таблице имеются повторяющиеся группы данных (информация о заказчиках). Эти данные необходимо вынести в отдельную таблицу.
- Столбец `ItemsOrdered` не содержит данные, которые по своему характеру были бы неразделимыми.

Для начала мы можем исключить из таблицы Orders несколько столбцов (табл. 8.3).

Таблица 8.3. Модифицированная таблица Orders

OrderNo (PK)	OrderDate	CustomerNo	ItemsOrdered
100	1/1/1999	54545	5-1A4536, Flange, 7lbs, $75; 4-OR2400, Injector, .5lbs, $108; 4-OR2403, Injector, .5lbs, $116; 1-4I5436, Head, 63lbs, $750
101	1/1/1999	12000	1-3X9567, Pump, 5lbs, $62.50
102	1/1/1999	66651	7-8G9200; Fan, 3lbs, $84; 1-8G5437, Fan, 3lbs, $15; 1-3H6250, Control, 5lbs, $32
103	1/2/1999	54545	40-8G9200, Fan, 3lbs, $480; 1-2P5523, Housing, 1lbs, $165; 1-3X9567, Pump, 5lbs, $42

После этого создадим из удаленных столбцов отдельную таблицу, Customers (табл. 8.4).

Таблица 8.4. Таблица Customers

CustomerNo (PK)	CustomerName	CustomerAddress
54545	ACME Co	1234 1st St.
12000	Sneed Corp.	555 Main Ave.
66651	ZZZ & Co.	4242 SW 2nd

В отношении старой таблицы и двух новых таблиц необходимо сделать несколько замечаний, приведенных ниже.

- Для новой таблицы Customers необходимо предусмотреть первичный ключ, чтобы обеспечить уникальность данных в каждой строке. Для таблицы Customers с данными о заказчиках могут быть предложены два потенциальных ключа — CustomerNo и CustomerName. Значения в столбце с идентификаторами заказчиков CustomerNo фактически были предназначены для выполнения именно этой задачи, поэтому выбор столбца CustomerNo для использования в качестве столбца первичного ключа кажется вполне обоснованным. Кроме того, нельзя исключить такую возможность, что в столбце CustomerName появится информация о заказчиках, компании которых имеют одинаковые названия. (Например, практика показала, что в США имеются сотни компаний с названием AA Auto Glass.)
- Безусловно, часть данных из таблицы Orders успешно изъята, но нам необходимо обеспечить возможность обращаться из таблицы Orders к данным новой таблицы Customers. Именно поэтому в таблице Orders по-прежнему находится столбец CustomerNo (столбец первичного ключа таблицы Customers). В дальнейшем, после формирования необходимых связей, будет создано ограничение внешнего ключа, позволяющее обеспечить наличие во всех заказах действительных номеров заказчиков.
- В результате создания таблицы Customers удалось исключить дублирующийся экземпляр информации о компании ACME Co. Тем самым продемонстрирован пример успешного решения задачи исключения данных, присутствующих в повторяющихся группах, для дальнейшего сохранения этих данных в единствен-

ном экземпляре. Благодаря этому, во-первых, удается сэкономить место, а во-вторых, избежать конфликтов, связанных с нарушением синхронизации при обновлении нескольких экземпляров повторяющихся данных.

- Изъятию подлежат только повторяющиеся группы данных, а не отдельные элементы данных. Например, в таблице может несколько раз повторяться одна и та же дата заказа, но такая дата в действительности не составляет группу; дата — это просто не связанный с другими фрагмент данных, который не имеет смысла за пределами таблицы.

Таким образом, выше было показано, какие действия следует предпринимать по отношению к повторяющимся данным. Теперь мы можем перейти к устранению второго нарушения требований к первой нормальной форме, чтобы добиться применения только неразрывных данных. В частности, изучение столбца `ItemsOrdered` показывает, что в нем фактически присутствует несколько различных фрагментов данных:

- данные о деталях, количество которых может составлять от одной и больше;
- информация о количестве, весе и стоимости отдельных деталей.

Отдельно взятые фрагменты данных о номере, количестве, весе, цене и стоимости деталей являются неразрывными, но теряют это свойство после их объединения в многозначные значения.

На первый взгляд это может показаться невероятным, но на практике некоторые приложения действительно организованы по такому же принципу. Дело в том, что при поверхностном подходе представляется очень удобным объединение всех взаимосвязанных данных в одном многозначном значении, ведь оформленные на бумаге счета-фактуры также представляют собой один крупный информационный блок, в котором имеются все данные о товарах, предназначенных для определенного заказчика, а это очень удобно. Поэтому иногда компьютеризированные системы создаются с учетом оформления данных по такому же принципу, который применяется для представления данных на бумаге.

Приступим к устранению этого недостатка и осуществим разбивку многозначного значения. В ходе этого обнаруживается, что целесообразно также ввести новый фрагмент информации, представляющий собой цену одной детали, или цену единицы товара (`UnitPrice`), как показано на рис. 8.1.

Order No (PK)	Order Date	Customer No	Part No	Description	Qty	Unit Price	Total Price	Wt.
100	1/1/1999	54545	1A4536	Flange	5	15	75	6
100	1/1/1999	54545	0R2400	Injector	4	27	108	.5
100	1/1/1999	54545	0R2403	Injector	4	29	116	.5
100	1/1/1999	54545	4I5436	Head	1	750	750	3
101	1/1/1999	12000	3X9567	Pump	1	62.50	62.50	5
102	1/1/1999	66651	8G9200	Fan	7	12	84	3
102	1/1/1999	66651	8G5437	Fan	1	15	15	3
102	1/1/1999	66651	3H6250	Control	1	32	32	5
103	1/2/1999	54545	8G9200	Fan	40	12	480	3
103	1/2/1999	54545	2P5523	Housing	1	165	165	1
103	1/2/1999	54545	3X9567	Pump	1	42	42	5

Рис. 8.1. Вариант таблицы `Orders`, в котором исключены многозначные значения, но не обеспечивается однозначная идентификация строк

Тем не менее после разбиения многозначного значения с данными о заказе на несколько фрагментов информации обнаруживается проблема, связанная с тем, что первичный ключ больше не может служить уникальным идентификатором для строк; несмотря на то, что в самих строках содержится уникальная информация, значения первичного ключа в некоторых из них повторяются.

Для устранения указанной проблемы могут применяться разные способы, но в данном примере просто введем в таблицу столбец с номерами строк заказа, как показано на рис. 8.2, чтобы снова получить возможность однозначно идентифицировать строки таблицы.

Order No (PK)	Line Item (PK)	Order Date	Customer No	Part No	Description	Qty	Unit Price	Total Price	Wt.
100	1	1/1/1999	54545	1A4536	Flange	5	15	75	6
100	2	1/1/1999	54545	0R2400	Injector	4	27	108	.5
100	3	1/1/1999	54545	0R2403	Injector	4	29	116	.5
100	4	1/1/1999	54545	4I5436	Head	1	750	750	3
101	1	1/1/1999	12000	3X9567	Pump	1	62.50	62.50	5
102	1	1/1/1999	66651	8G9200	Fan	7	12	84	3
102	2	1/1/1999	66651	8G5487	Fan	1	15	15	3
102	3	1/1/1999	66651	3H6250	Control	1	32	32	5
103	1	1/2/1999	54545	8G9200	Fan	40	12	480	3
103	2	1/2/1999	54545	2P5523	Housing	1	165	165	1
103	3	1/2/1999	54545	3X9567	Pump	1	42	42	5

Рис. 8.2. Вариант таблицы `Orders`*, обеспечивающий однозначную идентификацию строк*

> Вместо введения еще одного столбца, как в данном примере, можно было бы также применить другой подход — ввести в состав первичного ключа столбец с номерами деталей, `PartNo`. Но такое решение имеет один недостаток, связанный с тем, что стало бы невозможным включение в один и тот же заказ двух или нескольких строк с одинаковыми номерами деталей. Краткие сведения о ключах, основанных больше чем на одном столбце (называемых также составными ключами), будут приведены в следующей главе.

К этому моменту все требования к первой нормальной форме выполнены. Повторяющиеся группы данных отсутствуют, и значения во всех столбцах являются неразрывными (однозначными). Безусловно, возникают проблемы, связанные с тем, что некоторые данные приходится повторять в одном и том же столбце (поскольку все эти данные относятся к строкам с одинаковым значением первичного ключа), но решение данной проблемы будет вскоре описано.

Вторая нормальная форма

Следующий этап нормализации состоит в переходе ко второй нормальной форме (Second Normal Form — 2NF). Применение второй нормальной формы способствует дальнейшему сокращению количества повторяющихся данных (которые не должны обязательно составлять группы).

Вторая нормальная форма определяется в соответствии с двумя приведенными ниже правилами.

- Таблица должна соответствовать требованиям к первой нормальной форме (процесс нормализации является строго последовательным и напоминает кладку стены — в стену невозможно положить третий кирпич, не положив перед этим два первых).
- Каждый столбец должен зависеть от всего ключа.

В рассматриваемом примере как раз и обнаруживается нарушение требований ко второй нормальной форме (фактически даже несколько нарушений). Еще раз рассмотрим версию таблицы `Orders` в первой нормальной форме (см. рис. 8.2), чтобы определить, зависит ли каждый столбец от всего ключа, и нет ли таких столбцов, которые зависят только от части ключа.

Ответы на этих два вопроса являются соответственно отрицательным и положительным. В таблице `Orders` есть два столбца, которые зависят только от столбца `OrderNo`, но не от столбца `LineItem`. Таковыми являются столбцы `OrderDate` и `CustomerNo`: значения в обоих столбцах остаются одинаковыми для всего заказа, независимо от того, сколько отдельных позиций имеется в заказе. Для устранения указанного недостатка требуется ввести еще одну таблицу. При этом мы впервые сталкиваемся с противопоставлением таблиц заголовка и расшифровки.

Как показывает рассматриваемый пример, иногда на практике невозможно представить одну сущность с помощью одной таблицы и приходится разбивать одну таблицу на две; в связи с этим одна сущность разделяется на две сущности. Возникает связь двух таблиц, в которой таблица заголовка играет роль своего рода родительской таблицы по отношению к таблице расшифровки. При этом таблица заголовка хранит информацию, которую достаточно представить единожды, а таблица расшифровки содержит информацию, которая может существовать в нескольких экземплярах. При создании двух таких таблиц на основании исходной за таблицей заголовка обычно закрепляют имя исходной таблицы, а таблице расшифровки присваивают имя, которое содержит в начале имя таблицы заголовка и заканчивается подстрокой, указывающей, что эта таблица предназначена для описания содержимого заказов (например, `OrderDetails`; буквально — расшифровка заказов). К каждой отдельной строке таблицы заголовка, как правило, относится не меньше одной строки таблицы расшифровки, но, вообще говоря, таких строк может быть гораздо больше. В этом проявляется один из характерных примеров связи, которая будет рассматриваться в следующем крупном разделе (а именно, связи "один ко многим").

Итак, попытаемся устранить указанный недостаток и для этого еще раз разобьем одну из используемых таблиц (таблицу `Orders`). Фактически вначале целесообразно рассмотреть таблицу расшифровки, поскольку именно в ней находится основная часть оставшихся столбцов (табл. 8.5). Начиная с этого момента таблица расшифровки будет именоваться как `OrderDetails`.

Затем приступим к созданию таблицы заголовка (табл. 8.6). Хотя эта таблица после разбиения исходной таблицы на две кажется полностью новой, она предназначена для использования в качестве таблицы заголовка и поэтому сохраняет за собой имя `Orders`.

Итак, после этих преобразований требования ко второй нормальной форме выполнены. В частности, все столбцы зависят от всего ключа. Тем не менее читатель, скорее всего, не будет удивлен, узнав, что еще не все недостатки устранены. Речь об этом пойдет ниже.

Таблица 8.5. Таблица OrderDetails

OrderNo (PK)	LineItem (PK)	PartNo	Description	Qty	UnitPrice	TotalPrice	Wt
100	1	1A4536	Flange	5	15	75	6
100	2	OR2400	Injector	4	27	108	.5
100	3	OR2403	Injector	4	29	116	.5
100	4	4I5436	Head	1	750	750	3
101	1	3X9567	Pump	1	62.50	62.50	5
102	1	8G9200	Fan	7	12	84	3
102	2	8G5437	Fan	1	15	15	3
102	3	3H6250	Control	1	32	32	5
103	1	8G9200	Fan	40	12	480	3
103	2	2P5523	Housing	1	165	165	1
103	3	3X9567	Pump	1	42	42	5

Таблица 8.6. Таблица Orders

OrderNo (PK)	OrderDate	CustomerNo
100	1/1/1999	54545
101	1/1/1999	12000
102	1/1/1999	66651
103	1/2/1999	54545

Третья нормальная форма

Обычно после выполнения требований к третьей нормальной форме процесс нормализации заканчивается. С формальной точки зрения почти всегда остается возможность добиться еще большей степени нормализации по сравнению с этой, но, как правило, необходимые для этого процедуры не привлекают значительного интереса, кроме как в кругу теоретиков. Очень краткое описание следующих нормальных форм будет приведено ниже в данной главе, но вначале мы должны закончить свою основную работу по нормализации.

Как было указано в конце описания второй нормальной формы, созданная структура данных все еще характеризуется определенными недостатками, поскольку в ней еще не достигнута третья нормальная форма (Third Normal Form – 3NF). Применение третьей нормальной формы позволяет добиться того, чтобы ни один столбец в таблице не зависел от каких-либо других столбцов, кроме столбцов первичного ключа. Требования к данным, находящимся в третьей нормальной форме, перечислены ниже.

- Таблица должна соответствовать требованиям ко второй нормальной форме (как уже было сказано, переход от низших нормальных форм к высшим осуществляется строго последовательно).
- Ни в одном столбце не должны проявляться какие-либо зависимости от другого неключевого столбца.
- Наличие в таблице производных данных не допускается.

Уже известно, что таблицы в рассматриваемом примере находятся во второй нормальной форме, поэтому рассмотрим, насколько они соответствуют двум другим требованиям.

Вначале попытаемся выяснить, есть ли какие-либо столбцы, зависящие от столбца, отличного от столбца первичного ключа. Ответ на этот вопрос является положительным. Фактически в таблице `OrderDetails` имеется целый ряд столбцов, которые зависят от столбца `PartNo` не меньше, а, возможно, даже больше, чем от столбца первичного ключа рассматриваемой таблицы. В частности, от столбца `PartNo` полностью зависят столбцы `Weight` (Вес) и `Description` (Описание), поэтому необходимо снова выполнить разбиение одной таблицы на две.

На первый взгляд может показаться, что к этой категории относится также столбец `UnitPrice`, и это предположение действительно отчасти оправдывается. Безусловно, таблица `Products` с данными о товарах, которая будет создана для устранения указанного недостатка, может и должна содержать столбец `UnitPrice`, но сами данные этого столбца немного отличаются от других данных. В действительности дела обстоят так, что было бы лучше присвоить этому столбцу имя `ListPrice` (Цена по прейскуранту), поскольку цена на данный товар должна оставаться неизменной, независимо от того, в какой заказ включен указанный товар. Но включая в таблицу `OrderDetails` столбец `UnitPrice` (Цена за единицу товара) вместо столбца `ListPrice`, мы оставляем за собой две возможности. Во-первых, могут быть предложены скидки, с учетом которых цена товара в момент продажи изменится. Это означает, что цена, указанная в соответствующей строке таблицы `OrderDetails`, может отличаться от номинальной цены, указанной в таблице `Products`. Во-вторых, цена, которую мы можем назначить за единицу товара, может изменяться со временем под действием таких факторов, как инфляция, но будущие изменения цен не должны влиять на то, какие цены были включены в заказы, фактически выполненные в прошлом. Иными словами, цена – это одна из тех характеристик товара, которая в действительности имеет два аспекта, поскольку она зависит и от номера детали, `PartNo`, и от первичного ключа таблицы `OrderDetails` (иначе говоря, от значений `OrderID` и `LineItem`).

Прежде всего необходимо создать новую таблицу (назовем ее `Products`), предназначенную для хранения информации о деталях (в данном случае предметом торговли, т.е. товаром являются детали). В этой новой таблице будет храниться та информация, содержавшаяся в таблице `OrderDetails`, которая в большей степени зависит от значений в столбце `PartNo`, чем от значений в столбце `OrderID` или `LineItem` (табл. 8.7).

Таблица 8.7. Таблица `Products`

PartNo (PK)	Description	Wt
1A4536	Flange	6
OR2400	Injector	.5
OR2403	Injector	.5
4I5436	Head	3
3X9567	Pump	5
8G9200	Fan	3
8G5437	Fan	3
3H6250	Control	5
8G9200	Fan	3
2P5523	Housing	1
3X9567	Pump	5

После этого мы получаем возможность изъять из таблицы OrderDetails все указанные столбцы, кроме столбца внешнего ключа PartNo (табл. 8.8).

Таблица 8.8. Таблица OrderDetails

OrderNo (PK)	LineItem (PK)	PartNo	Qty	UnitPrice	TotalPrice
100	1	1A4536	5	15	75
100	2	OR2400	4	27	108
100	3	OR2403	4	29	116
100	4	4I5436	1	750	750
101	1	3X9567	1	62.50	62.50
102	1	8G9200	7	12	84
102	2	8G5437	1	15	15
102	3	3H6250	1	32	32
103	1	8G9200	40	12	480
103	2	2P5523	1	165	165
103	3	3X9567	1	42	42

В результате этого устраняется первый недостаток, препятствующий созданию третьей нормальной формы (зависимость между столбцами), но остается также необходимость исключить производные данные. В этом случае таблица OrderDetails содержит столбец TotalPrice, данные которого фактически могут быть получены путем перемножения значений, взятых из столбцов Qty UnitPrice. Наличие подобных столбцов полностью противоречит требованиям нормализации.

В настоящей книге автор иногда сознательно идет на нарушение требований нормализации, причем чаще всего такие нарушения допускаются применительно к производным данным. Это связано с тем, что применение производных данных позволяет существенно повысить быстродействие. Например, запрос с условием WHERE TotalPrice > $100 выполняется намного быстрее по сравнению с запросом, содержащим условие WHERE Qty * UnitPrice > 50, особенно если имеется возможность создать индекс на вычисленном столбце TotalPrice.

Но, с другой стороны, автор иногда применяет скорее гибридный подход, в котором используется вычисленный столбец, а сумма значений двух других столбцов поддерживается в актуальном состоянии с помощью СУБД SQL Server (напомним, что именно эта идея реализована в главе 5 в примере использования столбца PreviousSalary в таблице Employees базы данных Accounting). Если какой-то столбец становится очень важным с точки зрения производительности (например, если критерии, в которых учитываются значения из этого столбца, применяются во многих запросах), то может потребоваться задать индекс на этом новом вычисленном столбце. Важность индекса состоит в том, что в нем "материализуются" вычисленные данные, а это означает, что для СУБД SQL Server исключается необходимость рассчитывать значения в вычисленном столбце динамически. Вместо этого значение вычисляется только единожды, при сохранении данных о строке в индексе, а в дальнейшем используется заранее вычисленное значение столбца. Такая организация работы позволяет действительно добиться очень высокого быстродействия; дополнительная информация по этой теме приведена в главе 9.

Итак, чтобы выполнить все требования к созданию третьей нормальной формы, достаточно удалить столбец `TotalPrice` и в дальнейшем вычислять значения общей стоимости по мере необходимости.

Прочие нормальные формы

Кроме первых трех нормальных форм, разработаны еще три перечисленные ниже нормальные формы, которые рассматриваются в теоретических исследованиях как часть модели нормализации.

- Нормальная форма Бойса–Кодда (Boyce–Codd). Фактически эта нормальная форма считается разновидностью третьей нормальной формы. Она относится к той ситуации, в которой имеется несколько перекрывающихся потенциальных ключей. Такая ситуация может возникать только при следующих условиях:
  а) все потенциальные ключи являются составными ключами (иными словами, для создания ключа требуется больше одного столбца);
  б) количество потенциальных ключей больше одного;
  в) каждый потенциальный ключ имеет по меньшей мере один общий столбец с другим потенциальным ключом.

  Обычно для нормализации таблиц, имеющих указанные особенности, может применяться несколько разных способов, но на практике такая ситуация возникает крайне редко.
- Четвертая нормальная форма. Эта нормальная форма предназначена для устранения проблем, связанных с наличием многозначной зависимости. Многозначная зависимость возникает в такой ситуации, в которой ни один столбец в таблице не зависит от столбца, отличного от столбца первичного ключа, и все столбцы зависят от всего первичного ключа (т.е. таблица находится в третьей нормальной форме). Тем не менее при некотором довольно редком стечении обстоятельств может обнаруживаться отдельная зависимость одного из столбцов первичного ключа от других столбцов первичного ключа. Такая ситуация встречается редко и обычно не становится причиной каких-либо значительных проблем. Поэтому специалисты по базам данных чаще всего игнорируют возможность применения четвертой нормальной формы, и она не рассматривается в настоящей книге.
- Пятая нормальная форма. Определение этой нормальной формы касается вариантов проведения декомпозиции таблицы с потерями и без потерь. Дело в том, что иногда возникает необходимость разбить одну таблицу на несколько других таблиц, но при этом может сложиться такая ситуация, что по данным производных таблиц невозможно будет полностью восстановить данные исходной таблицы. Как и в случае других нормальных форм высокого порядка, ситуации, в которых может потребоваться применение пятой нормальной формы, встречаются редко, поэтому в настоящей книге эта нормальная форма больше не рассматривается.

Безусловно, приведенное выше описание нормальных форм высокого порядка сделано очень кратким, но автор принял решение об этом вполне осознанно. Дело в том, что в действительности эти формы приходится применять чрезвычайно редко, хотя и нужно иметь о них определенное представление.

Связи

Хотя связи между объектами, возникающие в реальной жизни, намного глубже и разностороннее по сравнению со связями между таблицами базы данных, изучение аналогий между теми и другими позволяет лучше разобраться в предмете.

В частности, по утверждению специалистов, ключом к успешному созданию связи является понимание того, какие роли играют оба участника связи, а также соблюдение допустимого регламента и правил поддержки связи. Под этим углом вполне могут рассматриваться и связи между объектами базы данных.

Связи, поддерживаемые между объектами в базе данных (в основном строками), подразделяются на три основных типа:

- "один к одному";
- "один ко многим";
- "многие ко многим".

Каждый из этих типов связей имеет определенные разновидности, определяемые тем, может ли одна из сторон связи быть пустой или нет. Например, иногда возникает необходимость ввести в действие вместо связи "один к одному" связь "нуль или один к одному".

Связь "один к одному"

Определение связи "один к одному" полностью соответствует ее названию. Связью "один к одному" называется такая связь, из наличия которой следует, что если имеется какая-то одна строка в одной таблице. то должна быть точно одна соответствующая ей строка в другой таблице.

В качестве примера связи "один к одному" рассмотрим вариант одного из фрагментов предыдущего примера. Предположим, что в крупной компании работой с заказчиками занимаются ее филиалы. Но компания, руководящая филиалами, должна иметь возможность следить за заказами всех заказчиков и подсчитывать суммарный итог покупок каждого заказчика, без учета того, в каком филиале (филиалах) сделаны эти покупки.

Допустим также, что во всех филиалах эксплуатируется одна СУБД, находящаяся в штаб-квартире головной компании, но в каждом филиале используется отдельная база данных. В таком случае для отслеживания информации обо всех заказчиках, что позволило бы в дальнейшем проще определять итоговые данные, целесообразно создать ведущую базу данных о заказчиках, принадлежащую головной компании. В таком случае филиалы могли бы сопровождать собственные таблицы заказчиков, поддерживая при этом связь "один к одному" с таблицей заказчиков головной компании. Это означает, что после ввода каждой строки с данными о заказчике в головной компании такая же строка должна появиться в одном из филиалов. С другой стороны, ввод строки с данными о заказчике в филиале должен повлечь за собой создание копии этой строки в головной компании.

Второй пример относится к ситуации, которая встречалась очень часто в версиях SQL Server, предшествующих 7.0. Речь идет о такой ситуации, в которой объем информации был слишком велик для того, чтобы его можно было поместить в одну строку. Напомним, что максимальный размер строки с данными, отличными от

BLOB, в СУБД SQL Server составляет 8060 байтов. Безусловно, эта величина намного больше по сравнению с 1962 байтами, которые служили в качестве ограничения размера строки в версии 6.5, но все равно возникают такие ситуации, в которых приходится хранить очень большое количество столбцов или пусть даже немного столбцов, но таких, которые содержат большой объем данных. Один из способов преодоления этого ограничения состоял в том, чтобы фактически создавались две разные таблицы и столбцы распределялись по этим таблицам. После этого между двумя подтаблицами устанавливалась связь "один к одному". Таким образом, благодаря применению соответствующих друг другу строк из двух таблиц появлялась возможность выполнить требования по хранению большого объема данных.

Непосредственно в самой СУБД SQL Server не предусмотрен полноценный способ поддержки связи "один к одному". Безусловно, может быть выдвинуто такое требование, что каждой строке таблицы `A` должна соответствовать определенная строка в таблице `B`, но после регламентации условий, согласно которым каждой строке таблицы `B` должна соответствовать определенная строка таблицы `A`, возникает неопределенная ситуация, связанная с тем, что становится неизвестным, в какую таблицу необходимо вводить строку в первую очередь. Если требуется обеспечить принудительную поддержку связи "один к одному" в СУБД SQL Server, то лучшее решение, которое может быть применено в этом случае, состоит в обеспечении принудительного выполнения всех операций вставки с помощью хранимой процедуры. А в этой хранимой процедуре должен быть реализован алгоритм, в соответствии с которым операция вставки осуществляется применительно к обеим таблицам или вообще не осуществляется. С подобной циклической зависимостью не могут справиться ни ограничения внешнего ключа, ни триггеры.

Связь "нуль или один к одному"

СУБД SQL Server обеспечивает поддержку экземпляров связи "нуль или один к одному". По существу такая связь аналогична связи "один к одному", за исключением того, что с одной из сторон связи допускается наличие или отсутствие строки, соответствующей другой строке.

Возвращаясь к примеру с головной компанией и филиалами, отметим, что в условиях данной задачи более целесообразно создать такую связь, в которой головная компания должна была бы иметь по одной строке, соответствующей строке с данными о заказчике из каждого филиала, но сами филиалы получали бы от головной компании только информацию о нужных им заказчиках. Например, может оказаться, что компания имеет филиалы, которые имеют дело с очень разными заказчиками (допустим, в одном филиале обслуживаются железнодорожные компании, а в другом строительные фирмы). В головной компании должна храниться информация обо всех заказчиках, независимо от того, чем они занимаются, но в филиале, обслуживающем строителей, вряд ли потребуется информация о железнодорожниках. В таком случае устанавливается связь, в которой сторона "нуль или один" соответствует филиалу, а сторона "один" — головной компании.

СУБД SQL Server позволяет обеспечить принудительную поддержку связей "нуль или один к одному" с помощью описанных ниже способов.

- С использованием сочетания уникального или первичного ключа с ограничением внешнего ключа. Ограничение внешнего ключа позволяет следить за тем,

чтобы существовала по меньшей мере одна строка в таблице на стороне "один" (в рассматриваемом примере – в таблице головной компании), но это ограничение не позволяет добиться того, чтобы существовала только одна строка (ограничение внешнего ключа выполняется, даже если в родительской таблице имеется несколько соответствующих ему строк). А применение первичного ключа или ограничения уникальности позволяет гарантировать соблюдение условия, согласно которому количество уникальных строк не превышает одного.

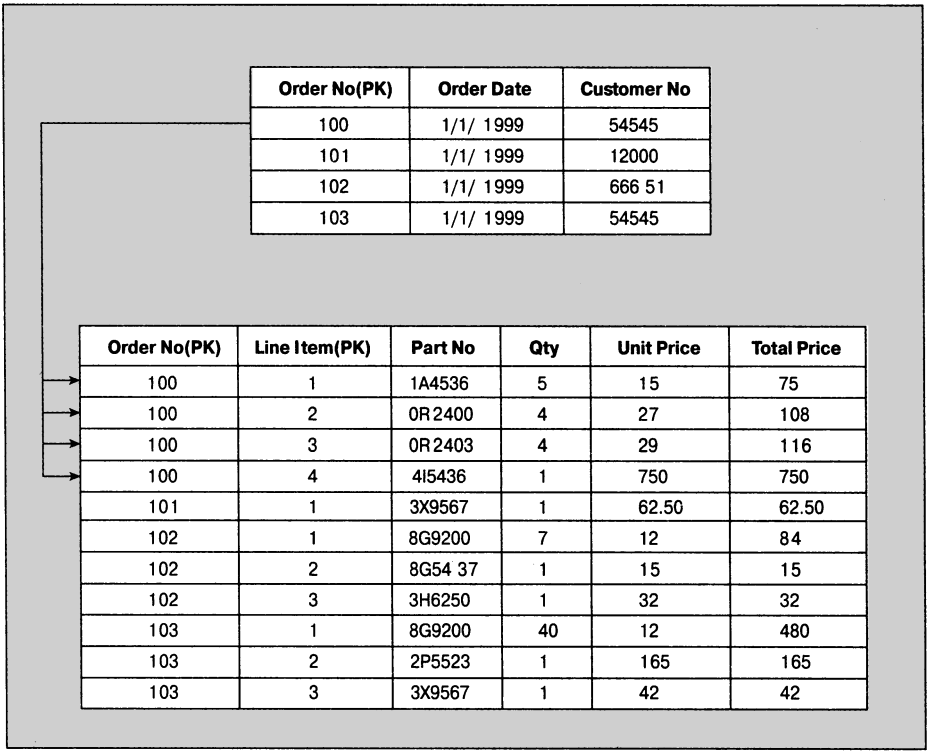
- С помощью триггеров. Следует отметить, что для поддержки связей "нуль или один к одному" триггеры должны быть заданы на обеих таблицах.

Причина, по которой в СУБД SQL Server легко обеспечивается поддержка связей "нуль или один к одному", но не связей "один к одному", заключается в том, что в последнем случае невозможно осуществить вставку строк одновременно в обе таблицы. Если связь полностью отвечает определению связи "один к одному", то невозможно выполнить вставку строки ни в одну из таблиц, поскольку в другой таблице еще нет соответствующей ей строки, поэтому возникает неразрешимая ситуация. С другой стороны, если применяется связь "нуль или один к одному", то может быть вначале выполнена вставка строки в таблицу на обязательной стороне связи (на стороне "один"), а затем, если потребуется, вставить вторую строку в таблицу на необязательной стороне (стороне "нуль или один"). Такая же проблема возникает и при использовании связи "один к одному или многим" и "один к нулю, одному или многим".

Связь "один к одному или многим"

Связь "один к одному или многим" представляет собой своего рода производственную, обычную, повседневно применяемую связь с помощью внешнего ключа. Обычно такая связь реализуется в виде определенной организации хранения данных на основе связей между таблицами заголовка и расшифровки. Превосходным примером такой связи может служить описанный выше вариант, в котором с помощью таблиц `Orders` и `OrderDetails` отдельно представлены заказы и содержимое заказов (рис. 8.3). Таблица расшифровки `OrderDetails` (находящаяся на стороне "один или многие" этой связи) не может применяться отдельно от таблицы заголовка, `Orders`, к которой она относится (безусловно, в таблице `OrderDetails` содержатся многие данные о заказе, но нет информации о том, кем сделан сам заказ). Аналогичным образом, не может отдельно использоваться и сама таблица `Orders`, поскольку в ней отсутствует информация о том, *что* фактически входит в сам заказ (в частности, информация, позволяющая выявлять такие ситуации, что для заказа зарезервирован номер, но сама расшифровка заказа не введена).

Тем не менее из-за наличия жесткой связи между таблицами возникает та же основная проблема, что и при использовании связей "один к одному". Дело в том, что снова приходится решать, в какую таблицу необходимо вставить строку (строки) в первую очередь. И в данном случае в СУБД SQL Server единственный способ полностью реализовать эту связь состоит в реализации требования, чтобы операции вставки или удаления всех данных осуществлялись с помощью хранимых процедур.



Order No(PK)	Order Date	Customer No
100	1/1/ 1999	54545
101	1/1/ 1999	12000
102	1/1/ 1999	666 51
103	1/1/ 1999	54545

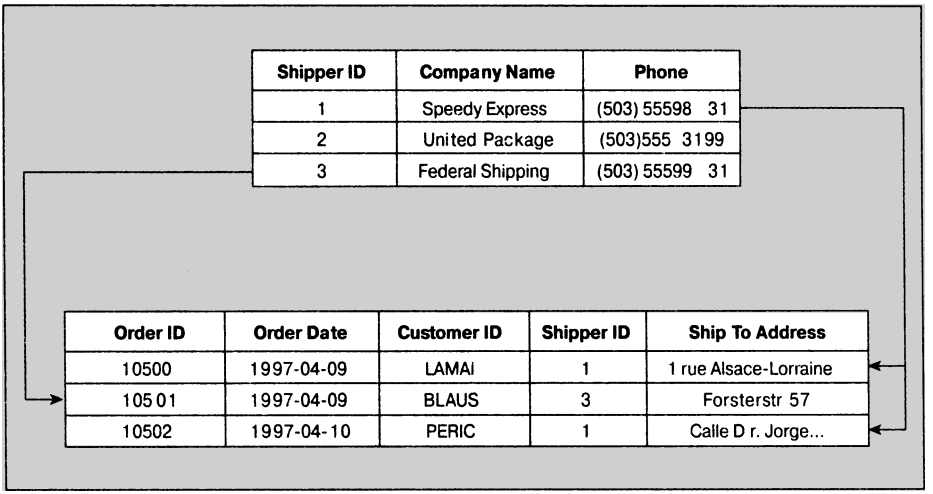
Order No(PK)	Line Item(PK)	Part No	Qty	Unit Price	Total Price
100	1	1A4536	5	15	75
100	2	0R2400	4	27	108
100	3	0R2403	4	29	116
100	4	4I5436	1	750	750
101	1	3X9567	1	62.50	62.50
102	1	8G9200	7	12	84
102	2	8G54 37	1	15	15
102	3	3H6250	1	32	32
103	1	8G9200	40	12	480
103	2	2P5523	1	165	165
103	3	3X9567	1	42	42

Рис. 8.3. Способ представления заказов и содержимого заказов с помощью таблиц `Orders` и `OrderDetails`

Связь "один к нулю, одному или многим"

Связь "один к нулю, одному или многим" — это еще одна разновидность производственной, обычной, повседневно применяемой связи с помощью внешнего ключа, которая, возможно, находит даже еще более широкое распространение, чем связь "один к одному или многим". Единственное заметное отличие в реализации этой связи по сравнению со связью "один к одному или многим" состоит в том, что допускается отсутствие значения в ссылающемся столбце (так называется столбец таблицы, на котором задано ограничение внешнего ключа); это означает, что из самого факта, что имеется строка в таблице на стороне "один", не обязательно следует наличие каких-либо экземпляров соответствующих ей строк в ссылающейся таблице (таблице на стороне "нуль, один или многие").

Пример реализации такой связи можно обнаружить в базе данных `Northwind` в виде связи между таблицами `Suppliers` и `Orders`. Таблица `Orders` позволяет следить за тем, какие поставщики занимаются доставкой заказанных товаров, но иногда товар забирает сам заказчик. Если же доставка осуществляется поставщиком, то необходимо обеспечить, чтобы в таблице `Suppliers` имелась информация обо всех поставщиках, занимающихся поставками. С другой стороны, как показано на рис. 8.4, вполне возможна такая ситуация, что в таблице `Orders` не найдут отражение данные обо всех без исключения поставщиках.



Shipper ID	Company Name	Phone
1	Speedy Express	(503) 55598 31
2	United Package	(503)555 3199
3	Federal Shipping	(503) 55599 31

Order ID	Order Date	Customer ID	Shipper ID	Ship To Address
10500	1997-04-09	LAMAI	1	1 rue Alsace-Lorraine
10501	1997-04-09	BLAUS	3	Forsterstr 57
10502	1997-04-10	PERIC	1	Calle D r. Jorge...

Рис. 8.4. Связь между таблицами **`Suppliers`** *и* **`Orders`**

Практически идентичный пример можно найти в базе данных `AdventureWorks` в виде связи между таблицами `Purchasing.PurchaseOrderHeader` и `Purchasing.ShipMethod`. Единственное реальное различие состоит в том, что в последнем случае рассматривается список компаний, занимающихся поставкой от заказчика, а не заказчику.

С помощью связи такого рода обычно задается так называемая *связь доменов*. *Доменом* называется ограниченный список значений, из которого должен осуществляться выбор при вставке данных в зависимую таблицу. При этом не допускается использование каких-либо данных, не находящихся в списке, который рассматривается как домен. Таблица, в которой хранятся строки с данными, образующими домен, обычно именуется *справочной*, или *поисковой* таблицей. По крайней мере одна справочная таблица обычно обнаруживается почти во всех применяемых на практике базах данных, а иногда количество таких таблиц довольно велико. В качестве справочной таблицы можно указать таблицу `Shippers`, которая не только обеспечивает хранение информации об идентификаторах и номерах телефонов поставщиков, но и регламентирует то, какие поставщики могут быть указаны в таблице `Orders`.

СУБД SQL Server позволяет обеспечить принудительную поддержку связи такого рода с помощью двух описанных ниже методов

- Ограничение внешнего ключа, `FOREIGN KEY`. Достаточно просто объявить ограничение `FOREIGN KEY`, распространяющееся на таблицу, которая используется в качестве стороны связи "многие", и сослаться на таблицу и столбец, который должен стать стороной "один" связи (наличие только одной строки в таблице, указанной в ссылке, гарантируется, поскольку на столбце (столбцах), указанном в объявлении внешнего ключа, должно быть задано ограничение `PRIMARY KEY` или `UNIQUE`).
- Триггеры. Фактически во всех ранних версиях SQL Server применение триггеров создавало единственную реальную возможность обеспечить ссылочную целостность. В действительности в базе данных необходимо предусмотреть два триггера, по одному для каждой стороны связи. Вначале триггер задается на

таблице, которая находится на стороне "многие", и с его помощью проверяется, имеет ли каждая вставляемая или обновляемая в этой таблице строка соответствие в той таблице, от которой она зависит (под этим подразумевается таблица на стороне "один" связи). После этого необходимо задать триггеры удаления и обновления на другой таблице. С помощью этих триггеров осуществляется проверка удаляемых и обновляемых строк в таблице, указанной в ссылке, для обеспечения того, чтобы в ссылающейся таблице не появились так называемые **висячие строки** (строки, которым не поставлена в соответствие ни одна строка в таблице домена).

В главе, посвященной ограничениям, уже затрагивался вопрос о том, как отличаются по своей производительности два описанных варианта. Как правило, вариант с использованием ограничения `FOREIGN KEY` позволяет добиться более высокого быстродействия, особенно в тех случаях, когда имеют место попытки нарушения ограничений. Несмотря на это, вариант с применением триггеров может все же оказаться наиболее приемлемым в тех ситуациях, когда требуется выполнить какое-то действие в связи с осуществлением каждой операции вставки или обновления данных (или требуется реализовать какие-то другие специальные ограничения).

Связь "многие ко многим"

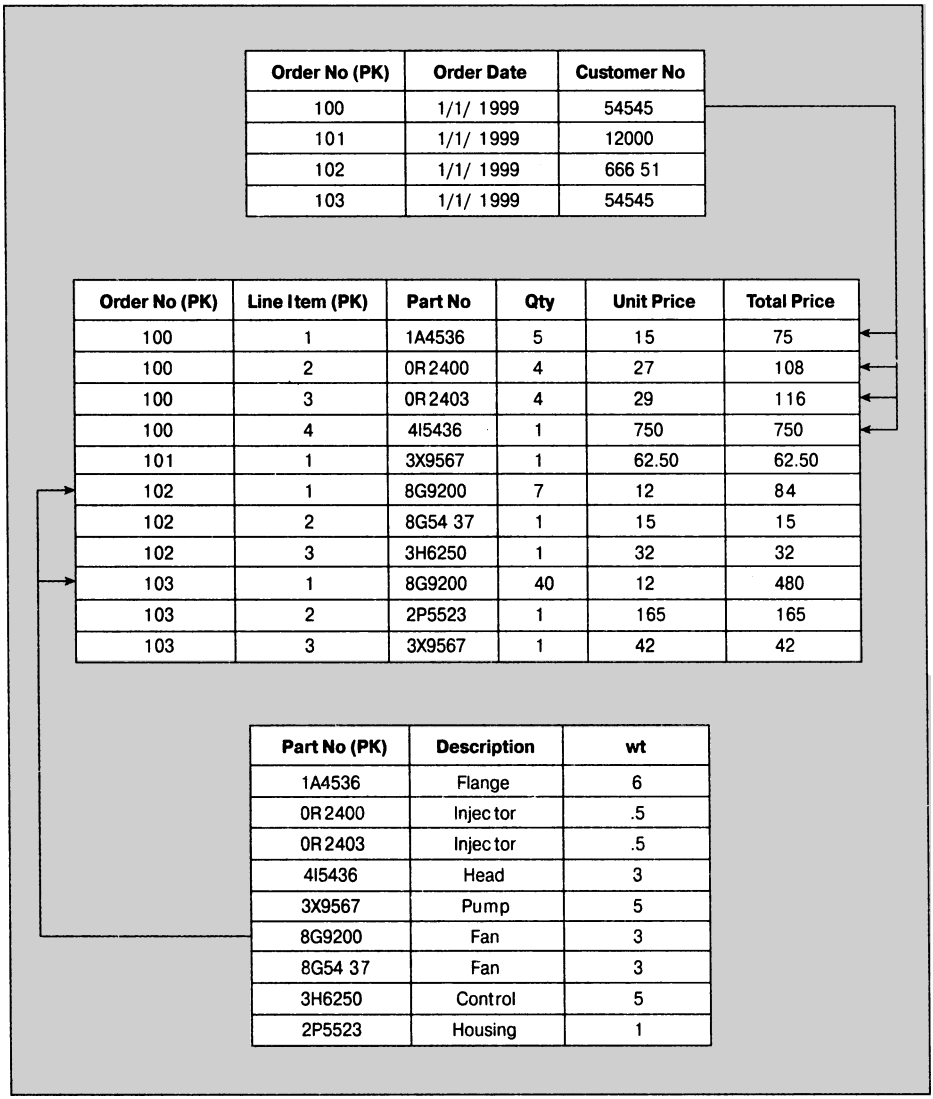
Связь "многие ко многим" характеризуется тем, что на обеих сторонах связи может присутствовать несколько согласующихся строк, а не только одна. В качестве примера такой связи можно указать связь между товарами и заказами. В каждом конкретном заказе может быть указано много разных товаров. С другой стороны, каждый конкретный товар может стать предметом нескольких разных заказов. Тем не менее все равно может потребоваться поддерживать связь между таблицами с данными о заказах и товарах, например, для обеспечения того, чтобы в заказах упоминались только имеющиеся в наличии товары (товары, данные о которых имеются в таблице `Products`).

В СУБД SQL Server не предусмотрен способ непосредственного определения связи "многие ко многим", поэтому для организации подобной связи применяется способ, основанный на использовании промежуточной таблицы. (В некоторых таблицах связи "многие ко многим" создаются почти случайно, в ходе обычного процесса нормализации, а в других таблицах создание подобной связи предусматривается с самого начала процесса проектирования базы данных с единственной целью — обеспечить взаимодействие между таблицами по такому принципу.) Промежуточная таблица, выполняющая роль "посредника", часто именуется *связующей* таблицей, *соединительной* таблицей, а иногда *объединяющей* таблицей.

Вначале рассмотрим связь "многие ко многим", которая создается в ходе обычного процесса нормализации. Пример такой связи можно обнаружить в таблице `OrderDetails` базы данных `Northwind`, с помощью которой создается связь "многие ко многим" между таблицами `Orders` и `Products` (рис. 8.5).

Операции соединения, которые рассматривались в главе 4, позволяют решить задачу определения того, в каких заказах упоминается какой-то конкретный товар, или выполнить обратную задачу — узнать, какие товары указаны в определенном заказе.

Перейдем ко второму варианту создания связи "многие ко многим", в котором специально создается с нуля соединительная таблица, позволяющая ввести в действие связь "многие ко многим". Рассмотрим в качестве примера связь между пользователями и группами прав доступа, которые могут иметь пользователи в системе.



Order No (PK)	Order Date	Customer No
100	1/1/ 1999	54545
101	1/1/ 1999	12000
102	1/1/ 1999	666 51
103	1/1/ 1999	54545

Order No (PK)	Line Item (PK)	Part No	Qty	Unit Price	Total Price
100	1	1A4536	5	15	75
100	2	0R2400	4	27	108
100	3	0R2403	4	29	116
100	4	4I5436	1	750	750
101	1	3X9567	1	62.50	62.50
102	1	8G9200	7	12	84
102	2	8G54 37	1	15	15
102	3	3H6250	1	32	32
103	1	8G9200	40	12	480
103	2	2P5523	1	165	165
103	3	3X9567	1	42	42

Part No (PK)	Description	wt
1A4536	Flange	6
0R2400	Injector	.5
0R2403	Injector	.5
4I5436	Head	3
3X9567	Pump	5
8G9200	Fan	3
8G54 37	Fan	3
3H6250	Control	5
2P5523	Housing	1

Рис. 8.5. Связь "многие ко многим" между таблицами `Orders` *и* `Products`

Начнем с определения таблицы прав доступа `Permissions`, которая может выглядеть так, как показано в табл. 8.9.

Таблица 8.9. Таблица `Permissions`

PermissionID	Description
1	Read
2	Insert
3	Update
4	Delete

Затем определим таблицу Users (табл. 8.10).

Таблица 8.10. Таблица Users

UserID	Full Name	Password	Active
JohnD	John Doe	Jfz9..nm3	1
SamS	Sam Spade	klk93)md	1

При такой организации данных возникает проблема, связанная с тем, как определить права доступа, предоставленные тем или другим пользователям. В первую очередь напрашивается такое решение, что в таблицу Users необходимо ввести столбец Permissions с идентификаторами прав доступа.

Таблица 8.11. Таблица Users со столбцом Permissions

UserID	Full Name	Password	Permissions	Active
JohnD	John Doe	Jfz9..nm3	1	1
SamS	Sam Spade	klk93)md	3	1

Но ошибочность этого решения сразу же становится очевидной, поскольку вновь созданная таблица не позволяет найти выход из такой ситуации, когда пользователям потребуется предоставить права на выполнение нескольких различных операций.

Если бы использовался обычный, плоский файл, то можно было бы просто записывать всю информацию о правах доступа в одно поле (табл. 8.12).

Таблица 8.12. Гипотетическая структура плоского файла с информацией о правах доступа

UserID	Full Name	Password	Permissions	Active
JohnD	John Doe	Jfz9..nm3	1,2,3	1
SamS	Sam Spade	klk93)md	1,2,3,4	1

Но файл со структурой, показанной в табл. 8.12, представляет собой пример нарушения требований к первой нормальной форме, согласно которым все значения во всех столбцах должны быть неразрывными (однозначными). Кроме того, применение такой структуры приводит к весьма значительному замедлению работы, поскольку после выборки каждого отдельного многозначного значения поля необходимо проводить его синтаксический анализ с помощью процедурных средств.

Дело в том, что фактически между этими двумя таблицами, Users и Permissions, существует связь "многие ко многим", поэтому требуется лишь найти способ представления такой связи в базе данных. Эту задачу можно решить, введя в действие соединительную таблицу, как показано на рис. 8.6. Еще раз отметим, что соединительная таблица чаще всего не вносит какие-либо новые данные в базу данных, а определяет ассоциацию между строками двух других таблиц.

После ввода в действие новой таблицы (назовем ее UserPermissions) появляется возможность назначать пользователям любые наборы прав доступа.

Следует отметить, что реализация требований к ссылочной целостности применительно к обеим таблицам является одинаковой, поскольку каждая из базовых таблиц (таблиц, которые содержат основополагающие данные и участвуют в связи "многие ко многим") имеет связь "один ко многим" с ассоциативной таблицей. Для обеспечения поддержки ограничений ссылочной целостности можно применить триггер или ограничение FOREIGN KEY.

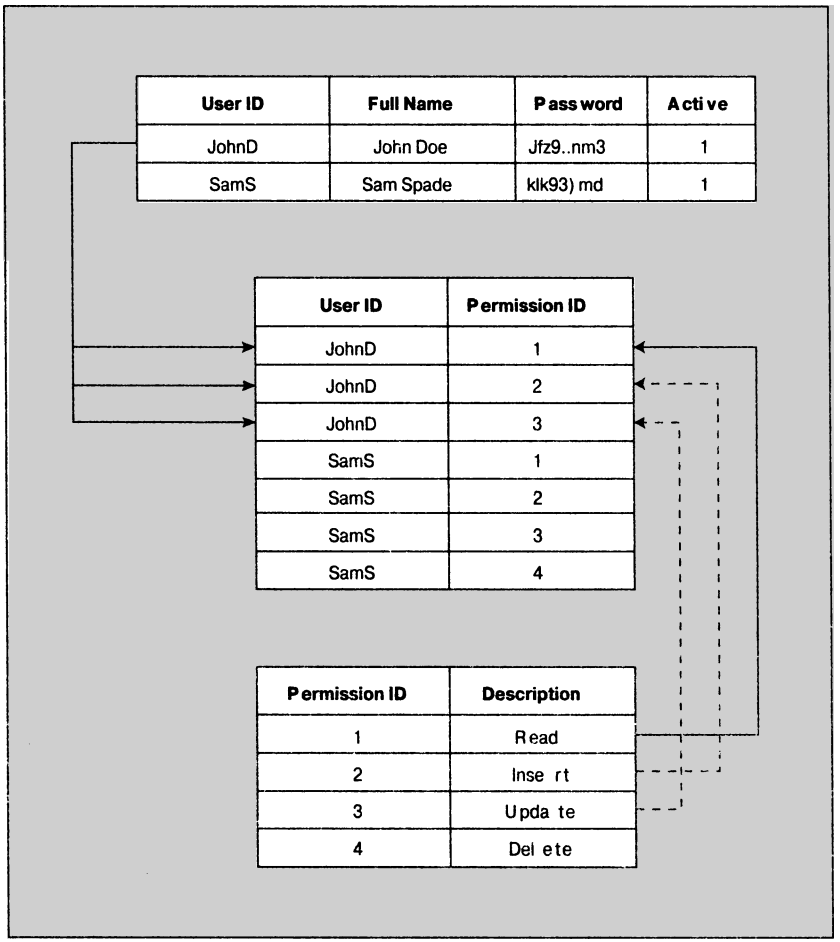


Рис. 8.6. Применение таблицы `UserPermissions` *для создания связи "многие ко многим" между таблицами* `Users` *и* `Permissions`

Средства построения диаграмм

Разработка качественного проекта базы данных без использования такого важного средства, как диаграммы "сущность–связь" (Entity Relationship Diagrams – ERD, или ER-диаграммы), почти не осуществима. Безусловно, небольшие базы данных обычно удается легко создать с помощью нескольких сценариев и непосредственно их реализовать, вообще не применяя каких-либо схем. Но чем больше становится база данных, тем сложнее удерживать все, что к ней относится, "в своей голове". ER-диаграммы позволяют устранять многочисленные проблемы, поскольку позволяют быстро представить визуально и уточнить все нюансы, касающиеся применяемых сущностей и их связей.

К счастью, в комплект программных средств SQL Server входит очень простое инструментальное средство построения диаграмм, которое может использоваться в качестве отправной точки при формировании простейших ER-диаграмм.

Прежде чем приступить к описанию этой темы, автор провел целый ряд дискуссий с заинтересованными лицами. Дело в том, что, с одной стороны, для решения ответственных задач построения ER-диаграмм, как правило, следует применять программы, специально предназначенные для этой цели (информация о некоторых программах такого типа приведена в приложении В.). Подобные инструментальные средства почти всегда поддерживают по меньшей мере один из нескольких методов построения диаграмм, широко применяемых на практике, а некоторые программы, в большей степени рассчитанные на массового пользователя, такие как Visio, обеспечивают возможность применения даже нескольких методологий формирования ER-диаграмм. Но, с другой стороны, инструментальное средство построения ER-диаграмм входит в состав программного обеспечения SQL Server, и проблема заключается именно в этом. В версии SQL Server 2005 применяется программа, основанная на использовании методологии построения диаграмм, которая использовалась корпорацией Microsoft в множестве инструментальных средств в течение многих лет. Но эта методология не соответствует ни одному из тех подходов к построению ER-диаграмм, которые широко применяются на практике. Тем не менее автор принял такое же решение, как и при написании всех других книг аналогичной тематики, и взял за основу инструментальные средства, которые входят в комплект программного обеспечения SQL Server. Однако читателю следует ознакомиться с коммерчески доступными инструментальными средствами построения ER-диаграмм, чтобы больше узнать о том, насколько велики их возможности и как их применение способствует уменьшению затрат труда на проектирование базы данных.

Чтобы открыть инструментальные средства, предусмотренные в СУБД SQL Server, достаточно перейти к узлу Diagrams, относящемуся к той базе данных, для которой требуется сформировать диаграмму (вначале разверните узел с обозначением сервера, а затем – узел базы данных). Некоторые действия, связанные с созданием диаграмм, которые будут описаны в данном разделе, покажутся читателю знакомыми, поскольку отдельные приведенные ниже диалоговые окна совпадают с теми, которые были описаны в главе 5 при создании таблиц.

Инструментальные средства построения диаграмм SQL Server не предоставляют слишком больших возможностей, поэтому усвоение данной темы, по-видимому, не потребует больших усилий. А в действительности для тех, кто знает программу редактирования связей СУБД Access, может показаться, что большая часть инструментальных средств SQL Server является для них знакомой.

Практическое занятие

Создание диаграммы

Приступим к созданию первой диаграммы. Для этого необходимо щелкнуть правой кнопкой мыши на узле Diagrams, находящемся под узлом базы данных `Northwind`, и выбрать опцию New Database Diagram.

Как было указано в главе 5, может появиться диалоговое окно (если предпринимается первая попытка создать диаграмму) с предостережением, согласно которому некоторые объекты, необходимые для обеспечения построения диаграмм, отсутствуют в базе данных, и с вопросом, следует ли создать эти объекты; в таком случае выберите положительный ответ, Yes.

Выполнение действий по созданию диаграммы в СУБД SQL Server начинается с развертывания такого же диалогового окна Add Table (рис. 8.7), которое было показано в главе 5; единственное различие между двумя окнами состоит в том, что в них перечислены разные таблицы.

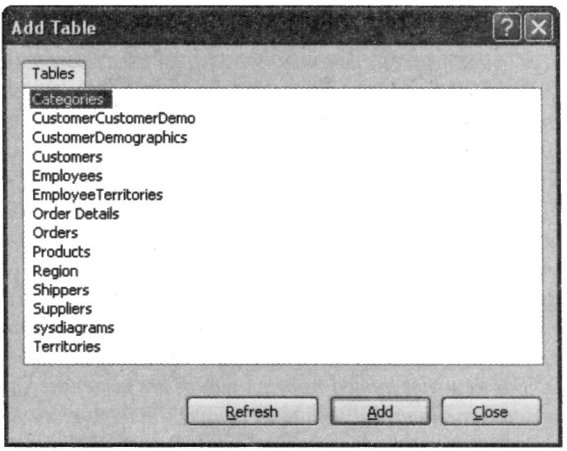


Рис. 8.7. Диалоговое окно Add Table

Выберите все таблицы (напомним, что, для того чтобы выделить подсветкой несколько таблиц, необходимо держать нажатой клавишу <Ctrl>), кроме таблицы `sysdiagrams` (следует отметить, что `sysdiagrams` – это фактически системная таблица, предназначенная исключительно для обеспечения возможности построения диаграмм), как показано на рис. 8.8.

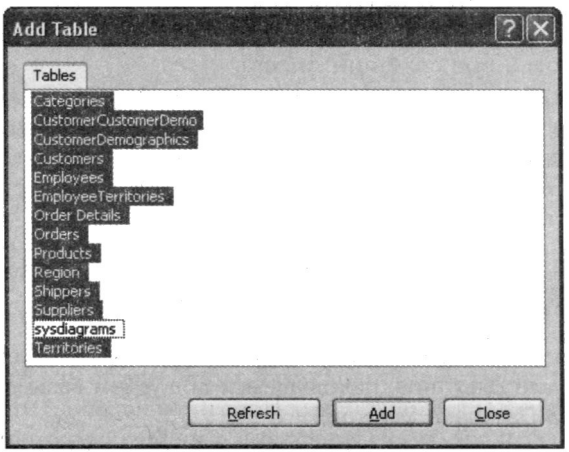


Рис. 8.8. Диалоговое окно Add Table, в котором выбраны все таблицы, кроме `sysdiagrams`

После того как вы щелкнете на кнопке Add, наступит короткая пауза, на то время, пока в программном обеспечении SQL Server будут формироваться изображения всех выбранных таблиц; щелкните на кнопке Close. Будет сформирована диаграмма, состоящая из всех указанных таблиц, но в зависимости от разрешающей способности экрана может оказаться, что диаграмму очень сложно рассмотреть из-за неподходящего коэффициента масштабирования диаграммы. Чтобы иметь возможность просматривать в окне больше таблиц, откорректируйте значение коэффициента масштабирования на панели инструментов. Для поиска подходящего компромиссного значения, которое позволило бы просматривать сразу большое количество таблиц и вместе с тем обеспечить разборчивость надписей, необходимо затратить определенные усилия, но всегда остается возможность подобрать такое значение коэффициента масштабирования, которое могло бы полностью соответствовать вашим личным требованиям. В данном примере автор выбрал значение 70%, чтобы можно было просматривать в окне сразу все таблицы, как показано на рис. 8.9 (но обычно применительно к базе данных, в которой имеется такое же значительное количество таблиц, какое, как правило, используется на практике, указанная цель недостижима).

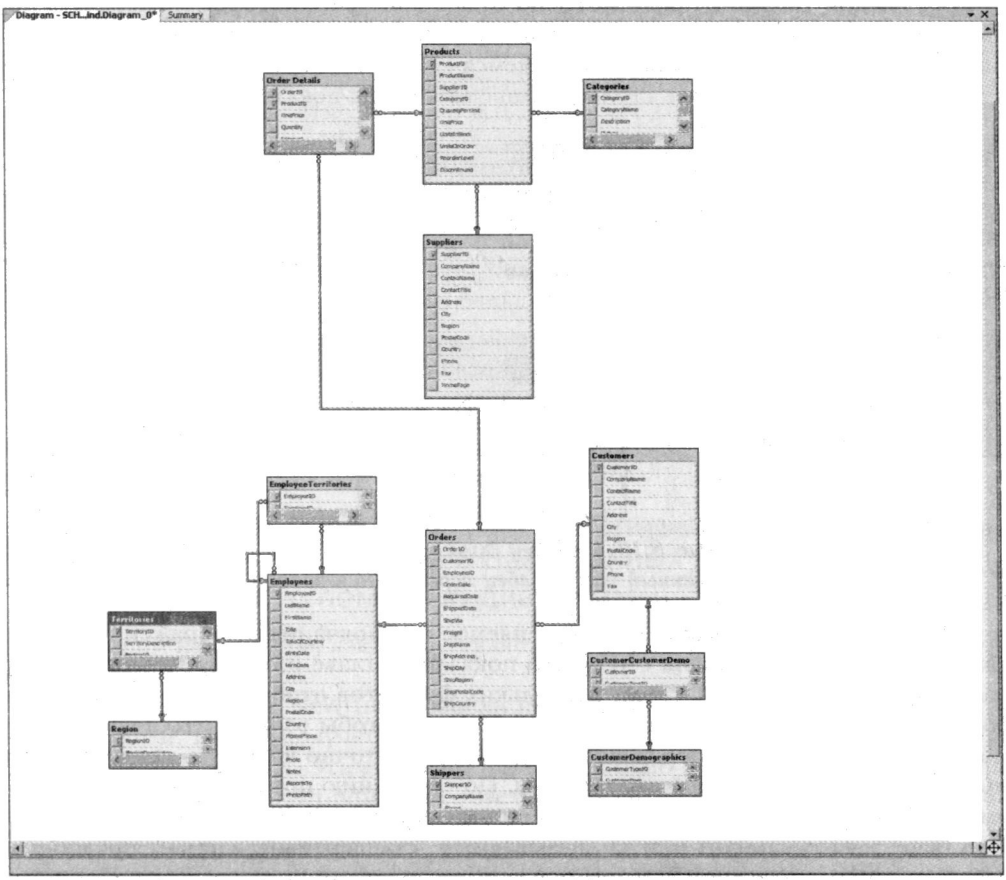


Рис. 8.9. Вид диаграммы, позволяющий просматривать сразу все таблицы

Описание полученных результатов

Очевидно, что на рис. 8.9 приведено намного больше информации по сравнению с теми диаграммами, которые создавались с помощью инструментальных средств, описанных в главе 5. С помощью программного обеспечения SQL Server была исследована каждая таблица, в отношении которой указано, что она должна быть представлена на диаграмме, после чего были проанализированы все прочие объекты, относящиеся к каждой из этих таблиц. Все прочие элементы изображения, приведенного на рис. 8.9, относятся к многочисленным объектам, соединяющим таблицы друг с другом, таким как первичные и внешние ключи.

Воспользуемся диаграммой на рис. 8.9 как отправной точкой для изучения того, как действует инструментальное средство построения диаграмм, и формирования некоторых новых таблиц.

Таблицы

Для каждой таблицы предусмотрено отдельное окно, которое может быть передвинуто в окне диаграммы. Первичный ключ изображается с помощью небольшой пиктограммы ключа в столбце, находящемся слева от поля с именем столбца; в качестве примера можно указать столбец `CustomerID`, как показано на рис. 8.10.

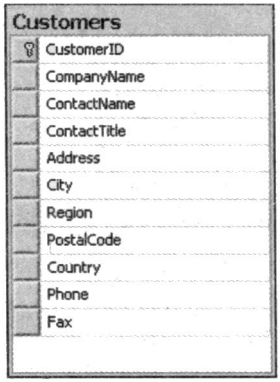


Рис. 8.10. Столбец `CustomerID`, обозначенный как столбец первичного ключа

В том, как сформировано это применяемое по умолчанию изображение таблицы, также есть полная аналогия с тем, что показано в главе 5, поскольку предусмотрена возможность выбирать среди нескольких форматов отображения, что позволяет корректировать любые характеристики таблицы. Чтобы ознакомиться со всеми возможными вариантами изображения таблицы, достаточно щелкнуть правой кнопкой мыши в окне интересующей вас таблицы. По умолчанию предусмотрен вывод только имен столбцов, но представляет интерес также вариант Custom, подобный описанному в главе 5; этот вариант или так называемый "стандартный" вариант представляет собой способ изображения, который позволяет модифицировать характеристики таблицы непосредственно с помощью диаграммы (а это очень удобно).

Добавление и удаление таблиц

Для добавления новой таблицы к диаграмме можно воспользоваться одним из двух описанных ниже способов.

Если решено включить в диаграмму таблицу, существующую в базе данных, но не представленную на диаграмме, достаточно щелкнуть на кнопке Add Table панели инструментов в окне программы формирования диаграмм. На экране появится список всех таблиц базы данных; после этого остается лишь выбрать ту таблицу, которую необходимо ввести в диаграмму, и на диаграмме появится изображение этой таблицы с указанием всех связей с другими таблицами.

Если же необходимо ввести в диаграмму еще не существующую, новую таблицу, щелкните на кнопке New Table панели инструментов в окне программы формирования диаграмм или щелкните правой кнопкой мыши в окне диаграммы и выберите команду New Table. Появится окно с приглашением ввести имя новой таблицы; после ввода имени таблица появится на диаграмме в представлении Column Properties. Затем достаточно отредактировать свойства таблицы, указав имена столбцов, типы данных и другие необходимые характеристики, после чего в базе данных появится новая таблица.

Следует отметить, что при выполнении указанных действий необходимо учитывать некоторые предостережения.

Прежде всего задайте в таблице первичный ключ. Программное обеспечение SQL Server не выполняет автоматически это действие и даже не выводит соответствующее приглашение для ввода данных о первичном ключе (как это предусмотрено в СУБД Access). Но сам процесс добавления первичного ключа является не совсем очевидным. Чтобы ввести первичный ключ, необходимо вначале выбрать столбцы, которые должны войти в состав этого ключа, а затем щелкнуть на них правой кнопкой мыши и выбрать команду Set Primary Key.

Кроме того, следует учитывать, что новая таблица фактически не создается в базе данных до тех пор, пока не будет выбрана команда Save. Такое же утверждение остается справедливым по отношению к любым операциям модификации характеристик, выполненным в ходе этой работы.

Практическое занятие

Создание в базе данных новой таблицы с помощью диаграммы

Рассмотрим на примере, как осуществляется создание в базе данных новой таблицы с помощью диаграммы.

Щелкните на кнопке New Table панели инструментов в окне программы построения диаграмм. После получения приглашения ввести имя задайте имя таблицы `CustomerNotes`. Вслед за этим должно появиться окно редактирования характеристик новой таблицы, в котором используется стандартное представление, Standard (рис. 8.11).

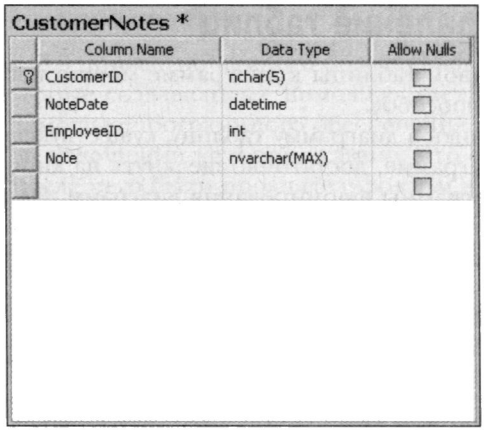


Рис. 8.11. Стандартное представление таблицы

На рис. 8.11 показано, что в определение таблицы уже введено несколько столбцов, а также задан первичный ключ (напомним, что для этого необходимо отметить столбцы, которые должны войти в состав первичного ключа, а затем щелкнуть на них правой кнопкой мыши и выбрать команду Set Primary Key). Прежде чем щелкнуть на кнопке Save, чтобы сохранить введенное определение таблицы, осуществите один эксперимент — откройте программу Management Studio и попытайтесь выполнить запрос к новой таблице:

```
SELECT * FROM CustomerNotes
```

При этом должно появиться следующее сообщение об ошибке:

```
Msg 208, Level 16, State 1, Line 1
Invalid object name 'CustomerNotes'.
```

Появление этого сообщения об ошибке вызвано тем, что рассматриваемая таблица существует только в виде отредактированного изображения на диаграмме; фактически таблица не создается вплоть до того момента, пока не будут сохранены внесенные изменения.

> Как показано на рис. 8.11, рядом с именем `CustomerNotes` этой таблицы в окне диаграммы справа от имени стоит знак звездочки (*). Наличие такого знака свидетельствует о том, что не все изменения, внесенные в характеристики таблицы, сохранены в базе данных.

Теперь снова переключитесь на программу Management Studio. В этой программе предусмотрены два описанных ниже варианта сохранения изменений в таблице.

- Команда Save. При использовании этой команды внесенные изменения отображаются на диаграмме и сохраняются в базе данных (указанная команда представлена на панели инструментов небольшой пиктограммой с изображением диска).
- Команда Save Change Script. С помощью этой команды изменения сохраняются в виде сценария, который может быть вызван на выполнение позднее. (Эту команду можно найти в меню Table Designer или задать ее как команду Save To

Text, если используется кнопка сохранения с изображением диска, о которой шла речь в предыдущем пункте.)

Перейдите к дальнейшим действиям и выберите команду Save, после чего появится приглашение дать подтверждение (дело в том, что после выполнения этой команды база данных изменится, а какого-либо способа отменить это действие не существует).

Подтвердите необходимость внести заданные изменения и попытайтесь снова вызвать указанный выше запрос на выполнение применительно к таблице `CustomerNotes`. На этот раз ошибка не должна возникнуть, поскольку необходимая таблица к этому времени уже создана (какие-либо строки с результатами не будут получены, но выполнение запроса должно завершиться успешно).

Описание полученных результатов

Во время построения диаграммы в программном обеспечении SQL Server незаметно для пользователя создается сценарий, который выглядит так же, как сценарии, создаваемые при оформлении внесенных изменений (см. главу 6). Но фактически код этих сценариев не вырабатывается и не выполняется до тех пор, пока не будет выбрана команда сохранить диаграмму.

Итак, таблица `CustomerNotes` успешно введена в базу данных, но предположим, что в ходе дальнейшей работы обнаруживается проблема, связанная с тем, что первичный ключ, применяемый в определении таблицы, позволяет сохранять только по одному примечанию, относящемуся к каждому заказчику. Однако практика показывает, что чаще всего со временем таких заметок, касающихся каждого заказчика, должно становиться все больше и больше. Это означает, что требуется изменить определение первичного ключа. Для решения указанной задачи можно применить несколько вариантов в зависимости от рассматриваемых требований, включая варианты, описанные ниже.

- Ввести в состав первичного ключа столбец с обозначением даты. Однако такое решение имеет два недостатка. Во-первых, возможность оставлять свои заметки должна быть предусмотрена для всех служащих, и может оказаться, что два сотрудника компании пожелают ввести свои замечания одновременно. Разумеется, этот недостаток несложно исправить, введя в состав первичного ключа еще один столбец, `EmployeeID`, который содержит идентификаторы служащих. Во-вторых, не исключено, что даже один и тот же служащий может принять решение ввести несколько полностью отдельных примечаний в один и тот же день (безусловно, если столбец с обозначением даты будет объявлен с типом данных `datetime`, то введенные в него значения практически всегда будут уникальными, поскольку невозможно добиться того, чтобы время выполнения двух различных операций ввода совпадало с точностью до миллисекунды, но данный пример рассматривается как чисто гипотетический). В таком случае даже введение столбца `EmployeeID` в состав первичного ключа не позволит обеспечить однозначную идентификацию строк.
- Ввести еще один столбец, чтобы усовершенствовать структуру первичного ключа. Этот метод можно было бы реализовать путем добавления столбца со счетчиком, с помощью которого нумеровались бы все заметки, касающиеся каждого отдельного заказчика. Еще один вариант может предусматривать

просто добавление столбца идентификации, гарантирующего уникальность. В результате этого первичный ключ уже фактически утрачивает свойство единственно применимого уникального идентификатора, но обычно мы от этого ничего не теряем (хотя и становимся вынужденными поддерживать больше одного индекса). Тем не менее такой подход позволяет создавать практически неограниченное количество заметок в расчете на каждого заказчика.

Сам автор намеревается применить подход, предусматривающий добавление к таблице столбца со счетчиком заметок, который будет именоваться `Sequence`. В соответствии с принятым соглашением (оно является необязательным, и не все разработчики его придерживаются) столбцы первичного ключа в изображении таблицы на бумаге обычно находятся на левом краю и задаются в объявлении таблицы в первую очередь. Если бы указанная задача усовершенствования структуры таблицы должна была быть решена в процессе эксплуатации базы данных, то, по-видимому, пришлось бы вызвать на выполнение оператор `ALTER TABLE` и добавить столбец. Но в таком случае новый столбец оказался бы в конце списка столбцов. Поэтому, чтобы исправить положение, потребовалось бы скопировать все данные в промежуточную таблицу, уничтожить все связи, в которых участвует старая таблица, уничтожить старую таблицу, создать новую таблицу, в которой уже предусмотрены все необходимые столбцы, а сами столбцы заданы в должном порядке, с помощью оператора `CREATE`, после чего восстановить все связи и скопировать данные из промежуточной таблицы в новую таблицу (очевидно, что это — продолжительный и утомительный процесс). Тем не менее, если используются инструментальные средства построения диаграмм, все эти действия выполняются программным обеспечением SQL Server.

Чтобы вставить новый столбец после любого столбца в таблице, достаточно щелкнуть правой кнопкой мыши на определении столбца, который непосредственно следует за вставляемым столбцом. Как показано на рис. 8.12, инструментальное средство построения диаграмм сдвигает определения всех последующих столбцов вниз, чтобы освободить место для нового столбца.

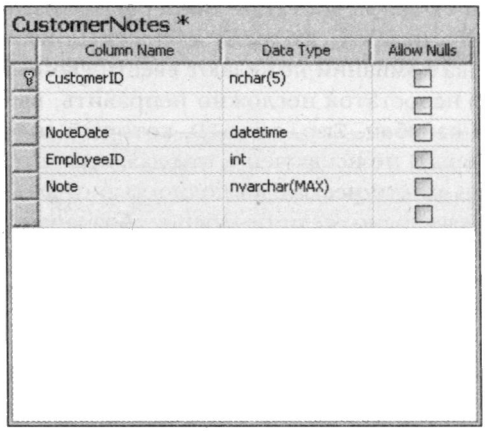


Рис. 8.12. Результат подготовки к вставке нового столбца перед существующими столбцами таблицы

После этого можно ввести определение нового столбца, как показано на рис. 8.13, и переопределить первичный ключ (выбрать определения обоих столбцов, `CustomerID` и `Sequence`, щелкнуть на выделенном участке правой кнопкой мыши и выбрать команду Set Primary Key).

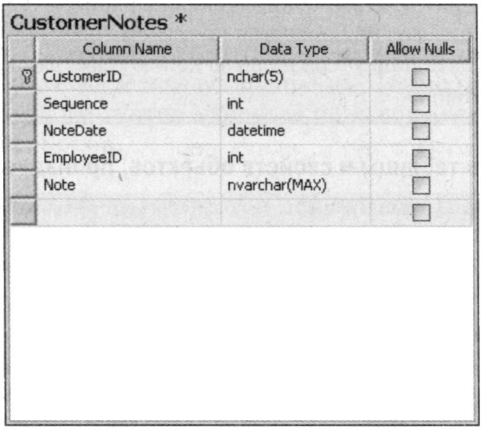


Рис. 8.13. Окно таблицы `CustomerNotes` после ввода определения нового столбца

После этого следует щелкнуть на кнопке Save, чтобы сформировать таблицу, в которой предусмотрен требуемый порядок столбцов. Исключительно для того, чтобы убедиться в этом, попробуем воспользоваться процедурой `sp_help`:

```
EXEC sp_help CustomerNotes
```

В выводе этой процедуры будет показано, что столбцы действительно заданы в должном порядке:

```
....
CustomerID
Sequence
NoteDate
EmployeeID
Note
....
```

Среди всех инструментальных средств формирования диаграмм, применяемых для выполнения таких операций, как изменение порядка столбцов, особо выделяется программа daVinci. Автору приходилось сталкиваться со многими другими инструментальными средствами ER-диаграмм, причем все они предоставляли возможность синхронизации изменений последовательности столбцов между базой данных и диаграммой, однако в действительности результаты не были столь однозначными (иными словами, следует очень тщательно подходить к выбору инструмента, применяемого для внесения изменений в реальные данные). Безусловно, характеристики всех инструментальных средств неизменно улучшаются, но программа daVinci по-прежнему носит отпечаток гениальности, присущей тому человеку, чье имя легло в ее название.

Кроме того, необходимо сделать еще одно предостережение, касающееся данной темы. Если приходится работать с реальными данными, то желательно использовать вариант внесения изменений, в основе которого лежат сценарии, но не следует вносить изменения с помощью оперативного соединения с базой данных. Дело в том, что при такой организации работы обеспечивается возможность полностью проверить работу сценария на экспериментальных базах данных, не подвергая ненужному риску реальные данные. Следует также обязательно создать полную резервную копию базы данных и только после этого вносить подобные изменения.

Редактирование свойств таблицы и свойств объектов, принадлежащих к таблице

Кроме основных атрибутов таблиц, которые рассматривались до сих пор, операции редактирования можно также применять ко многим другим характеристикам таблицы. Для доступа к соответствующим характеристикам и последующего редактирования или добавления используются два способа, описанных ниже.

- Свойства. Для редактирования свойств применяется окно, которое всплывает и по умолчанию прикрепляется в правой части интерфейса Management Studio в окне построения диаграмм. Чтобы вызвать на экран окно свойств, Properties, щелкните на пиктограмме с обозначением Properties Window панели инструментов в программе Management Studio.
- Объекты, принадлежащие к таблице, такие как индексы, ограничения и связи. Свойства этих объектов редактируются в отдельных диалоговых окнах, доступ к которым можно получить, щелкнув правой кнопкой мыши на изображении таблицы на диаграмме и выбрав элемент, свойство которого необходимо отредактировать.

В процессе редактирования свойств объектов с помощью диаграмм необходимо учитывать некоторые важные требования, поэтому прежде всего рассмотрим основные диалоговые окна, применяемые в этом процессе.

Окно Properties

Окно Properties, относящееся к таблице `CustomerNotes`, показано на рис. 8.14.

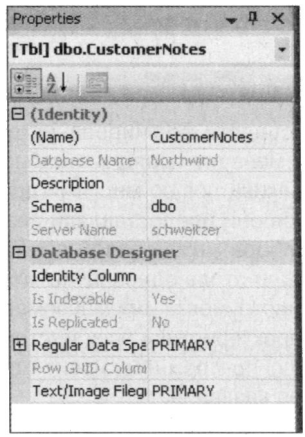


Рис. 8.14. Окно Properties таблицы **`CustomerNotes`**

Окно свойств, Properties, позволяет задавать некоторые важные свойства таблицы. Наиболее существенными из них являются свойство, позволяющее узнать, к какой схеме относится таблица, а также свойство, с помощью которого можно определить, имеет ли таблица столбец идентификации, `Identity`.

Окна Relationships

Диалоговые окна Relationships, в соответствии со своим названием, позволяют регламентировать характер связей между таблицами. Пример такого окна показан на рис. 8.15. Вполне очевидно, что в этом окне еще не заданы какие-либо связи, относящиеся к таблице `CustomerNotes`.

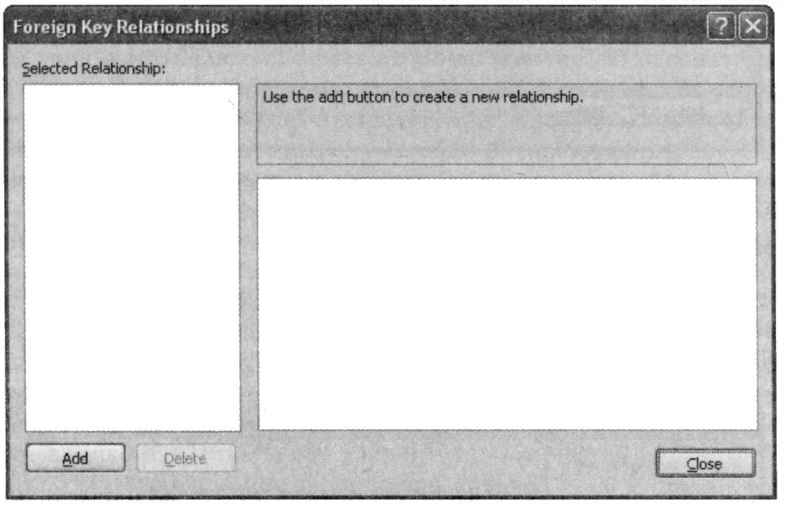


Рис. 8.15. Диалоговое окно Relationships

На данный момент отметим лишь то, что с помощью этого окна можно редактировать практически любые характеристики, которые касаются свойств связей. Например, чтобы создать связь с другой таблицей, достаточно щелкнуть на кнопке Add и задать значения различных полей. Эта тема будет рассматриваться более подробно ниже в данной главе.

Окно Indexes/Keys

Описание этой темы может показаться читателю немного непонятным, поскольку в предыдущих главах еще не встречались сведения об индексах. Поэтому предварительно рассмотрим, какие данные, касающиеся ключей и индексов, приведены в окне Indexes/Keys (рис. 8.16).

Вполне очевидно, что окно, показанное на рис. 8.16, позволяет создавать, редактировать и удалять индексы. С помощью этого окна можно также указывать файловые группы, применяемые для хранения индексов (однако мы не советуем брать на себя такую ответственность). Дополнительные сведения об индексах приведены в следующей главе.

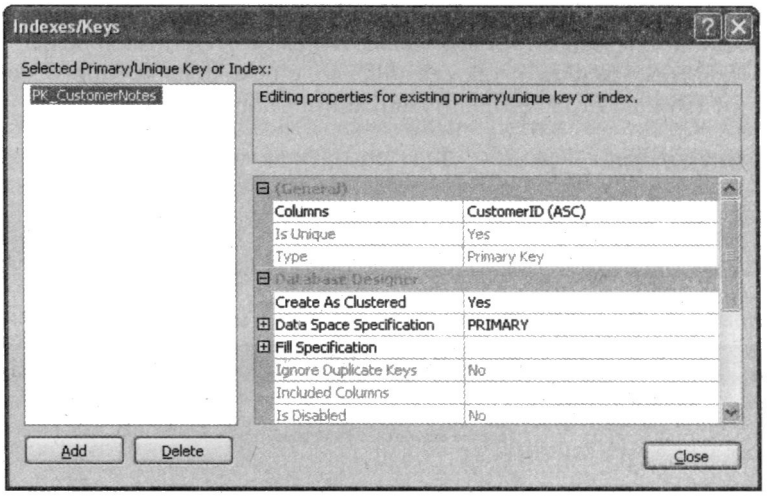


Рис. 8.16. Окно Indexes/Keys

Окно Check Constraints

Окно `Check Constraints` (рис. 8.17), в соответствии с его названием, позволяет задавать только ограничения проверки (для работы с ограничениями внешних ключей и значениями, применяемыми по умолчанию, предназначены другие окна).

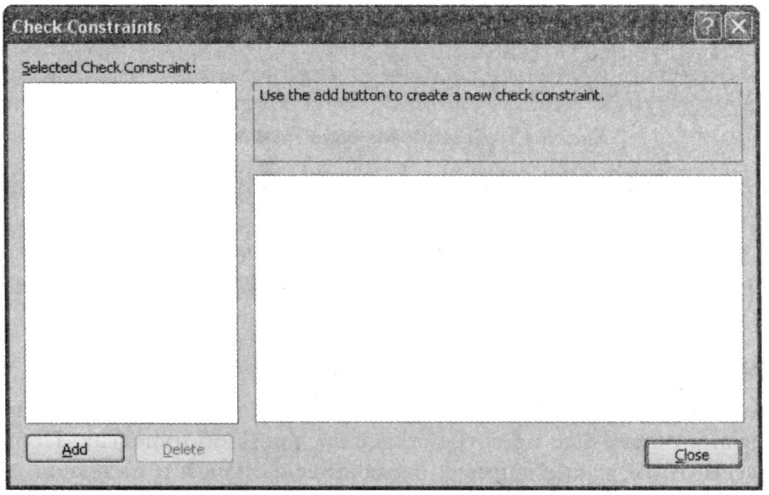


Рис. 8.17. Окно Check Constraints

Это окно, как и окно, показанное на рис. 8.15, остается пока не заполненным. Дело в том, что для таблицы `CustomerNotes` не определены какие-либо другие ограничения, кроме первичного ключа, а для работы с первичным ключом применяется окно Indexes/Keys. Еще раз отметим, что окно Check Constraints предназначено только для определения ограничений `CHECK`. Для ознакомления со всеми возможностями этого окна щелкните на кнопке Add и задайте соответствующее ограничение.

Дополнительные сведения об использовании окон Relationships

Выше было приведено краткое описание того, какие возможности это инструментальное средство построения диаграмм предоставляет применительно к таблицам. А в этом разделе, как уже было сказано, дополнительно описаны возможности указанного средства для определения связей (и рассматриваются некоторые подробности, касающиеся связей).

На диаграммах "сущность–связь", которые отображаются в окне инструментального средства построения диаграмм, связи между таблицами обозначены линиями. Та сторона связи, которая обозначена пиктограммой с изображением ключа, представляет собой сторону "один" рассматриваемой связи, а сторона, обозначенная символом бесконечности, соответствует стороне "многие". В рассматриваемом инструментальном средстве не предусмотрена возможность использовать линии связи, позволяющие представлять связи, характеризующиеся тем, что в них возможна нулевая кратностей связи (для связей с нулевой кратностью применяется такое же обозначение, как и для связи с единичной кратностью). Кроме того, на диаграмме фактически отображаются только такие связи, которые объявлены с использованием ограничений внешнего ключа. Объявление любой связи, принудительно поддерживаемой с помощью триггеров (независимо от того, к какому типу относится эта связь), не приводит к появлению на диаграмме линии, обозначающей связь.

Еще раз рассмотрим диаграмму базы данных `Northwind`. Для того чтобы подробнее ознакомиться с особенностями окна Relationships, щелкните правой кнопкой мыши на таблице `Customers` или `Orders` и выберите команду Relationships. На экране появится вариант диалогового окна Relationships с большим количеством заполненных полей по сравнению с тем вариантом, который рассматривался в предыдущем разделе. Диалоговое окно Relationships, относящееся к таблице `Orders`, показано на рис. 8.18.

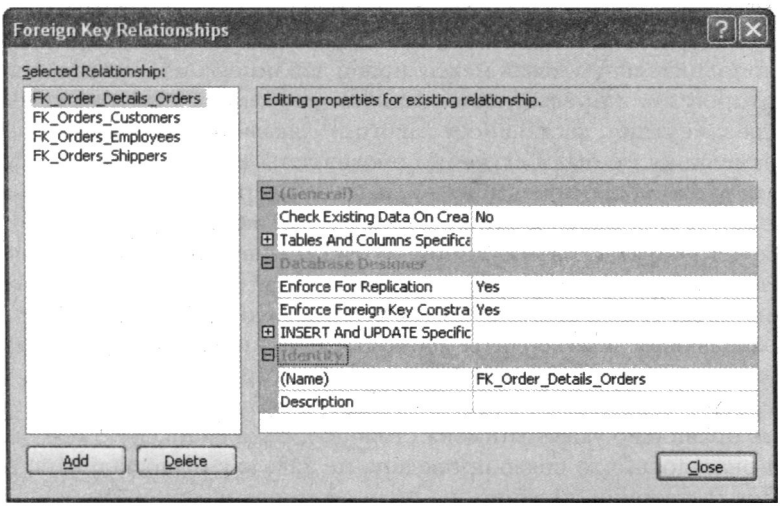


Рис. 8.18. Диалоговое окно Relationships для таблицы **`Orders`**

Окно, показанное на рис. 8.18, позволяет корректировать многие свойства связи, в частности, определять возможность выполнения каскадных действий; указывать, введены ли в действие или отменены ограничения внешнего ключа (например, если необходимо преднамеренно ввести такие данные, в которых нарушаются условия применения связи); а также корректировать имя связи.

Проектировщики баз данных еще не пришли к общему мнению в отношении того, какими должны быть правила именования связей, а некоторые из них вообще не придают особого значения тому, какие имена должны присваиваться связям. Сам автор предпочитает использовать для описания связей краткие предложения с глаголом. Например, для представления связи между таблицами `Customers` и `Orders` я считаю целесообразным применить имя `CustomerHasOrders` (Заказчик имеет заказы) или нечто подобное. Присваивание связи того или другого имени не является такой уж ответственной операцией (чаще всего имена связей даже не приходится использовать в приложении), но я пришел к выводу, что применение содержательных имен связей может оказаться действительно удобным, если приходится просматривать длинные списки объектов или изучать особенно сложную ER-диаграмму, на которой линии связей могут пересекать всю страницу диаграммы, минуя сущности, не охваченные связями.

Ввод дополнительных связей с помощью инструментального средства построения диаграмм

Задача определения новой связи является чрезвычайно простой; достаточно лишь перетащить линию связи и опустить ее на диаграмму. Единственная сложность заключается в том, что операция перетаскивания должна начинаться и заканчиваться именно в тех местах, которые были для этого намечены. Если у вас нет уверенности в том, что вы сможете успешно провести линию связи, выберите интересующий вас столбец (столбцы), прежде чем выполнять перетаскивание.

Практическое занятие

Формирование связи

Введем дополнительную связь между новой таблицей `CustomerNotes` (описание создания которой см. в предыдущем разделе) и таблицей `Customers`. (И действительно, вводя замечания, касающиеся какого-то заказчика, необходимо обеспечить, чтобы эти замечания не были случайно обозначены как относящиеся к другому заказчику.) Для этого щелкните кнопкой мыши на сером прямоугольнике слева от столбца `CustomerID` в обозначении таблицы `Customers` и удерживайте кнопку нажатой, после чего перетаскивайте линию, которая тянется за курсором мыши, до тех пор, пока эта линия не остановится на столбце `CustomerID` в определении таблицы `CustomerNotes`. После этого появится всплывающее диалоговое окно Tables and Columns, позволяющее подтвердить, действительно ли должна быть установлена такая связь между столбцами соответствующих таблиц (рис. 8.19).

Если операция перетаскивания и фиксации выполнена правильно, то сразу же должны быть правильно указаны имена столбцов, находящихся по обе стороны связи. Но если оказалось, что связь проведена не так, как было задумано, не следует беспокоиться; достаточно щелкнуть на поле со списком, относящемся к таблице, в которой необходимо выбрать не этот, а другой столбец, после чего указать новый столбец. Затем откорректируйте имя связи, указав вместо предусмотренного по умол-

чанию имени FK_CustomerNotes_Customers имя CustomerHasNotes. После щелчка на кнопке OK появится уже более привычное диалоговое окно с определением связи. С помощью этого диалогового окна можно определить все прочие параметры настройки, которые могут потребовать корректировки, а затем сохранить информацию о новой связи, относящуюся к текущей таблице. Займитесь этим и внесите изменения в определения действий, выполняемых в случае удаления и обновления (Delete Rule и Update Rule), указав значение Cascade. Это позволяет обеспечить, что в случае обновления или удаления строки с данными о заказчике будет также происходить по мере необходимости обновление или удаление строк с примечаниями, относящимися к этому заказчику. Описанные выше изменения показаны на рис. 8.20.

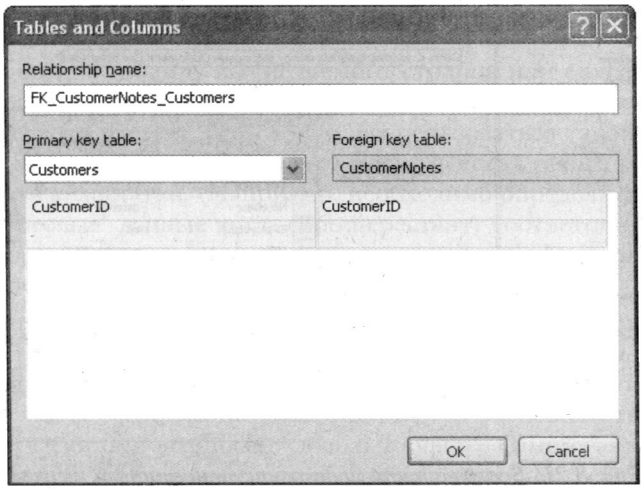


Рис. 8.19. Диалоговое окно Tables and Columns

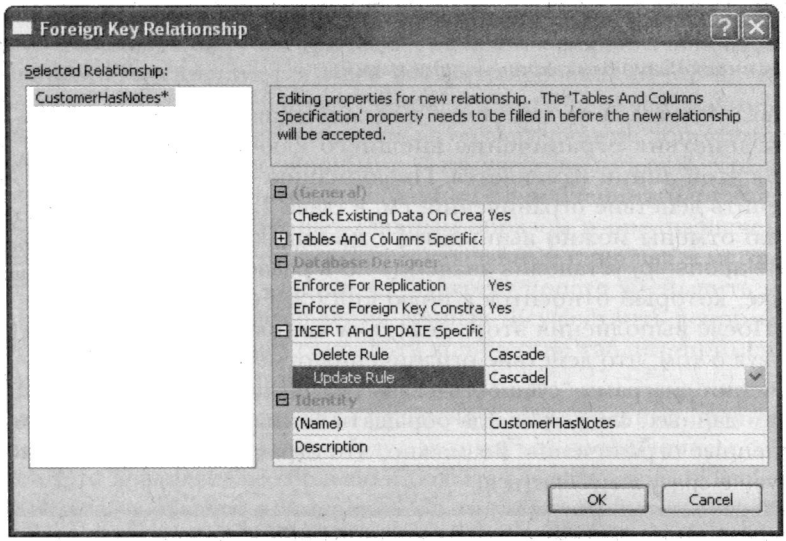


Рис. 8.20. Диалоговое окно с определением связи

Закончив выполнение указанных действий, щелкните на кнопке OK и ознакомьтесь с новой линией связи, появившейся на диаграмме.

Описание полученных результатов

Во многом аналогично тому, что происходит при выполнении операции добавления таблицы, описанной в предыдущем разделе, при внесении указанных изменений в программном обеспечении SQL Server формируется сценарий SQL. В текущем практическом занятии рассматривалось только создание на диаграмме новой связи. Перетащив курсор мыши над этой связью, можно обнаружить всплывающую подсказку, которая содержит имя связи и указывает, к чему она относится (рис. 8.21).

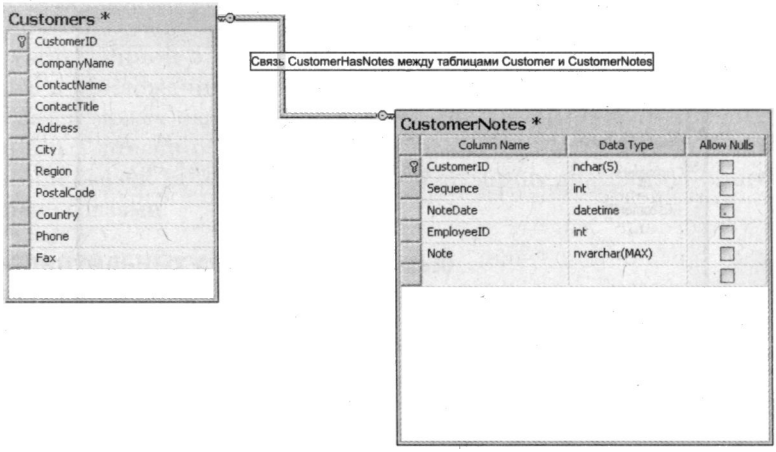


Рис. 8.21. Всплывающая подсказка, относящаяся к связи

Обратите внимание на то, что на рис. 8.21 рядом с именами обеих таблиц показаны звездочки. Дело в том, что внесенные изменения были лишь добавлены к списку изменений, относящемуся к данной диаграмме, а в физическую базу данных они будут внесены только после щелчка на кнопке Save *для сохранения диаграммы*

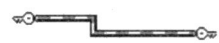


Рис. 8.22. Линия связи, соответствующая отмененному ограничению

Линия связи показана на диаграмме как сплошная, но в случае отмены действия ограничения внешнего ключа способ отображения этой линии изменяется. Информацию о том, как можно отменить действие ограничений, см. в главе 7. Но такую же операцию отмены можно выполнить с помощью диалогового окна Relationship, установив значение no в разворачивающемся списке, который относится к полю Enforce Foreign Key Constraint. После выполнения этого действия линия связи изменяется (рис. 8.22), свидетельствуя о том, что действие ограничения отменено.

При изучении диаграмм "сущность–связь", относящихся к фактически эксплуатируемым базам данных, следует всегда обращать внимание на линии связи, обозначающие отмененные ограничения. Возможно, что ограничение отменено не случайно, но в этом лучше сразу же убедиться.

Денормализация

Денормализация – это процесс, противоположный нормализации. Хотя задача денормализации является достаточно сложной, в настоящем разделе описанию этой темы будет отведено довольно мало места, однако прежде всего следует отметить, что иногда денормализация действительно является оправданной, но это не означает, что нужно отказаться от нормализации данных.

Как было указано выше в настоящей главе, иногда проектировщики баз данных слишком злоупотребляют нормализацией данных. Для них целью становится решение самой этой задачи как таковой, поэтому нормализация данных осуществляется исключительно ради нормализации, а не для усовершенствования структуры базы данных. Ниже приведен ряд соображений по этому поводу.

- Если включение в таблицу вычисляемого столбца или сохранение в ней некоторых производных данных позволяет повысить эффективность формирования отчетов, то следует, безусловно, воспользоваться такой возможностью. Но при этом не следует забывать, что преимущества такого решения должны перевешивать недостатки. (Например, необходимо определить, не случится ли так, что "итоговые" данные когда-либо перестанут соответствовать данным, на основе которых они получены. Как узнать, что этого не произошло, и как исправить положение, если такое случилось?)
- Иногда благодаря включению в таблицу лишь одного денормализованного столбца (или нескольких таких столбцов) удается избежать необходимости применения соединений для выборки информации или значительно сократить количество таких соединений. Нужно научиться распознавать подобные ситуации; фактически они возникают довольно часто. Самому автору встречались такие ситуации, в которых в результате добавления одного столбца к часто применяемой базовой таблице появилась возможность перейти от использования соединения девяти таблиц к соединению всего лишь трех таблиц и таким образом сократить продолжительность выполнения запроса примерно на 90%.
- Если ведется накопление исторических данных, которые в основном остаются неизменными и используются лишь для формирования отчетов, то проблемы обеспечения целостности данных становятся гораздо менее значимыми. После того как данные проверены и записаны в область памяти, допускающую только чтение, можно на полном основании быть уверенным в том, что при работе с этими данными не придется сталкиваться с проблемами нарушения синхронизации, для решения которых главным образом и предназначены методы нормализации данных. С того момента как данные становятся историческими, может оказаться гораздо удобнее (и быстрее) просто уменьшить количество уровней в структуре представления данных (осуществить денормализацию). Это позволяет значительно сократить количество таблиц, применяемых для хранения данных, и ускорить работу с ними.
- Чем меньше таблиц приходится применять для создания соединений, тем охотнее рядовые пользователи будут заниматься составлением собственных отчетов. Это позволяет все больше и больше улучшать подготовку сообщества пользователей в области применения тех инструментальных средств, которые для них предназначены. В конечном итоге начинает увеличиваться количество пользователей, обращающихся к администратору базы данных за разрешением

на предоставление непосредственного доступа к базе данных в целях получения собственных специализированных отчетов. Дело в том, что для многих пользователей база данных с высокой степенью нормализации выглядит как лабиринт и становится фактически неприменимой, а после некоторой денормализации данных работа основной массы сотрудников предприятия значительно упрощается.

Несмотря на все сказанное в этом разделе, если есть основания полагать, что из-за низкой степени нормализации база данных будет иметь низкую производительность, то следует обеспечить ее нормализацию, ведь реляционные системы создаются именно с этой целью. А если применяемая степень нормализации является слишком высокой, это лишь означает, что улучшается поддержка целостности данных и достигается высокая производительность в транзакционной среде.

Методы повышения производительности, не связанные с нормализацией

В настоящем разделе приведены основные рекомендации по проектированию, позволяющие повысить производительность базы данных, которые не связаны с применением нормализации. Лишь немногие из этих рекомендаций могут рассматриваться как четкие и однозначные правила. Скорее их можно трактовать как информацию к размышлению. При этом наиболее важное соображение заключается в том, что, несмотря на столь важную роль нормализации с точки зрения проектирования базы данных, это — не единственный метод усовершенствования структуры базы данных.

Неуклонное стремление к упрощению

Безусловно, время от времени разработчики в результате неустанных поисков обнаруживают новые, более эффективные способы организации работы. Некоторые выдвигаемые при этом идеи действительно являются невероятно восхитительными и чрезвычайно полезными. Встречаются и другие открытия, которые также великолепны, но не очень полезны. Однако не менее часто возникают предложения, которые, возможно, и носят отпечаток новизны, но не становятся практически применимыми лишь благодаря этому.

По мнению автора, не нужно подавлять в себе творческое начало и стремиться всегда использовать лишь давно сложившиеся методы. Не следует только забывать, что тщательно проверенные и признанные правильными подходы применяются снова и снова по одной причине — они обычно оправдываются.

В версии SQL Server 2005 предложено очень много мощных и гибких новых дополнений. Эта версия программного продукта позволяет реализовывать сложные правила работы с данными и даже сложные типы данных (с помощью управляемых кодом функций и типов данных) Поэтому стало еще труднее удержаться от соблазна увлечься новинками и реализовать на их основе более сложное решение по сравнению с минимально необходимым. Старайтесь избегать использования таких подходов, в результате осуществления которых база данных становится сложнее по сравнению с тем, что действительно требуется. Минималистский подход обычно (хотя и не всегда) приводит к созданию базы данных, которой не только проще управлять, но проще добиться от нее высокой производительности.

Правильный выбор типов данных

Основным принципом выбора наиболее подходящего типа данных является такая рекомендация, что следует всегда выбирать минимально допустимый тип данных. Иными словами, выбирайте необходимые типы данных, но только самые необходимые.

Например, если в базе данных требуется хранить данные о месяцах (представленные в виде числа от 1 до 12), то для этого достаточно предусмотреть один байт, воспользовавшись типом данных `tinyint`. Тем не менее автору регулярно приходится встречать такие базы данных, в которых поле, предназначенное для хранения только номера месяца, объявляется с типом данных `int` (для которого требуется 4 байта). Не используйте типы данных `nchar` и `nvarchar`, если вам никогда не потребуется обрабатывать данные, содержащие символы Unicode; для представления этих типов данных требуется в два раза больше байтов по сравнению с их аналогами, `char` и `varchar`, не предназначенными для поддержки кодировки Unicode.

Многие считают, что применение типов данных со слишком большим форматом представления приводит лишь к избыточному расходованию пространства. Обсуждая этот вопрос с другими специалистами, автор иногда слышит такое возражение: "Нужно ли об этом беспокоиться. Ведь в наши дни дисковое пространство стало таким недорогим!" Однако, не считая того, что жесткие диски SCSI качественных моделей все еще стоят так дорого, что нелегко решиться на их покупку, существует также проблема пропускной способности сети. Если при передаче информации каждой строки приходится передавать лишние 100 байтов, то обработка 100 строк приводит к тому, что через сеть приходится пропускать примерно 10 лишних килобайтов. Если эти цифры недостаточно убедительны, отметим, что в случае, если 100 пользователей проводят по 50 транзакций в час, то за это время при указанных условиях бесполезно расходуется пропускная способность сети на 50 Мбайт.

Необходимо также помнить, что подавляющее большинство операций обработки данных выполняется в базе данных многократно, поэтому даже небольшие упущения накапливаются как снежный ком и могут вызвать достаточно заметные нарушения в работе.

Сохранение максимально возможного объема накопленных данных

При проектировании базы вопрос о том, как долго должны храниться полученные ранее данные, не менее важен, чем все другие вопросы организации хранения данных.

Если база данных предназначена для накопления данных, то рано или поздно придется поинтересоваться тем, потребуется ли когда-либо еще та или иная информация. По мнению автора, в этом случае любые сомнения должны рассматриваться как свидетельство в пользу того, что не нужно спешить удалять данные. Дело в том, что данные, изъятые из базы данных и уничтоженные, чаще всего не могут быть восстановлены.

Можно гарантировать, что по крайней мере однажды (а также, возможно, еще много и много раз, но по другому поводу) в информационный отдел обратится заказчик (напомним, что заказчиками являются все, кому требуются услуги, поэтому заказчиком следует считать не только тех, кто обращается в компанию за платными услугами, но и представителей своей компании, которые просят предоставить им помощь) и выскажет примерно такое пожелание: "Можете ли вы составить отчет с

информацией о том, какие суммы денег были выплачены каждой сторонней акционерной компании в прошлом году?"

Но в связи с этим возникает вопрос о том, хранятся ли в базе данных такие сведения о каждом поставщике, по которым можно определить, что поставщик – акционерная компания. В таком случае лучше всех обстоят дела у тех компаний, которые работают в рамках налогового законодательства США (предоставляют форму отчетности 1099). Итак, у вас такие данные имеются, и вы даете положительный ответ на вопрос заказчика. Однако за этим следует очередная просьба: "Можете ли вы распечатать эти данные с указанием адреса поставщика, по которому он был зарегистрирован на конец прошлого года?"

Увы, можно не сомневаться в том, что в базе данных не хранятся адреса, которые использовались в прошлом, или, по крайней мере, не хранится дата внесения изменений в адрес. Короче говоря, невозможно предугадать заранее, какая информация, относящаяся ко всему периоду эксплуатации базы данных, когда-либо потребуется пользователю системы, и в этом несложно убедиться на практике. Из этого следует, что рекомендации по выбору минимально необходимых типов данных (приведенные выше) позволяют также затрачивать меньше ресурсов на обслуживание данных, изымаемых из повседневной эксплуатации. А если часть данных предназначена только для архивного хранения, то необходимо убедиться в том, что к ним невозможно получить бесконтрольный доступ с помощью любого из операторов `SELECT` постоянно эксплуатируемого приложения, если в этом нет необходимости (фактически аналогичный подход должен распространяться на все данные, независимо от того, для чего они предназначены).

> Если есть основания полагать, что в связи с выбором любого способа действий (т.е. в связи с тем, будет ли решено хранить данные за прошлый период или избавиться от них) могут наступить правовые последствия, то проконсультируйтесь со своим адвокатом. Иногда по закону требуется хранить данные в течение определенного времени, а в других обстоятельствах лучше сразу же удалять ненужные данные, как только подойдет срок, разрешающий это сделать законным путем.

Пример осуществления процедуры нормализации

Кратко рассмотрим пример осуществления процесса проектирования базы данных, применяемой для выставления счетов, описание которого уже фактически было начато в разделе, посвященном нормализации. При описании этой темы инструментальные средства построения диаграмм в основном будут применяться для изучения готового проекта, но в ходе этого затрагиваются также некоторые новые проблемы в целях демонстрации того, как их решение отражается на создаваемом проекте.

Создание базы данных

В отличие от многих инструментальных средств построения диаграмм, предлагаемых независимыми разработчиками, подобные инструментальные средства, входящие в состав программного обеспечения SQL Server, не позволяют создавать базы

данных, поэтому, чтобы иметь возможность воспользоваться диаграммами для работы с базой данных, необходимо вначале самому создать требуемую базу данных.

В рассматриваемом примере проблемы обработки данных не затрагиваются, поэтому достаточно создать небольшую базу данных. Назовем эту базу данных `Invoice`. Для этого воспользуемся соответствующим диалоговым окном программы Management Studio.

Щелкните правой кнопкой мыши на узле Databases под обозначением используемого сервера, выберите команду New Database и введите информацию о базе данных `Invoice`, которая приобретает размер 3 Мбайт.

В настоящей книге процесс создания базы данных уже рассматривался достаточно подробно, поэтому в целях сокращения объема изложения отметим, что в данном случае достаточно просто принять предусмотренные по умолчанию значения других параметров (рис. 8.23).

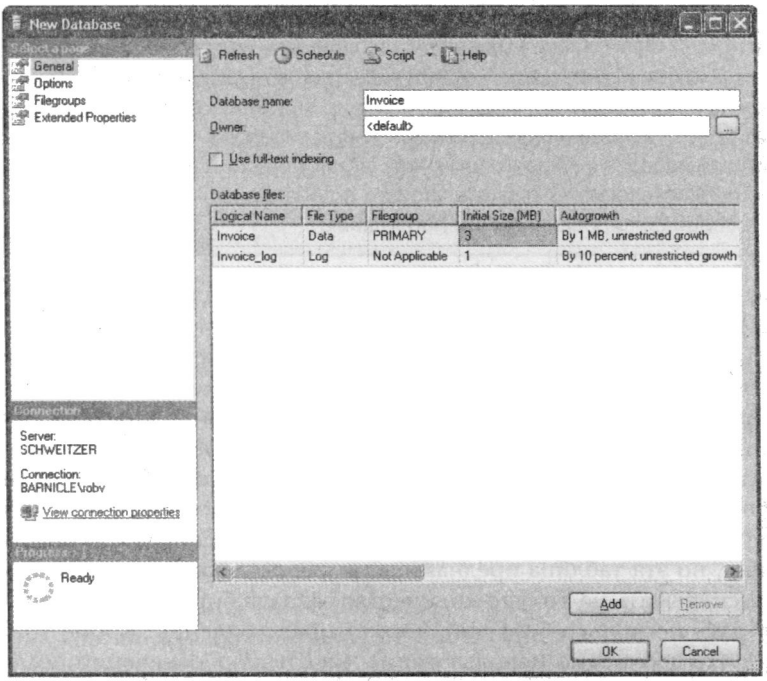


Рис. 8.23. Диалоговое окно New Database

Развертывание диаграммы и создание исходных таблиц

Как и при создании диаграммы для базы данных `Northwind`, разверните узел, относящийся к используемой базе данных (он должен появиться под узлом Databases), и выберите утвердительный ответ в диалоговом окне с запросом о том, нужно ли вводить в действие все объекты, необходимые для поддержки работы с диаграммами. Затем щелкните правой кнопкой мыши на узле Diagrams и выберите команду New Database Diagram. Появится всплывающее диалоговое окно Add Table, но в базе данных `Invoice` нет пользовательских таблиц, поэтому остается только щелкнуть на кнопке Cancel, чтобы открылось пустое окно.

Теперь мы можем приступить к добавлению новых таблиц. Для этого можно либо щелкнуть на пиктограмме New table панели инструментов, либо щелкнуть правой кнопкой мыши в любом месте в окне диаграммы и выбрать команду New Table. Начнем с добавления таблицы `Orders`, как показано на рис. 8.24.

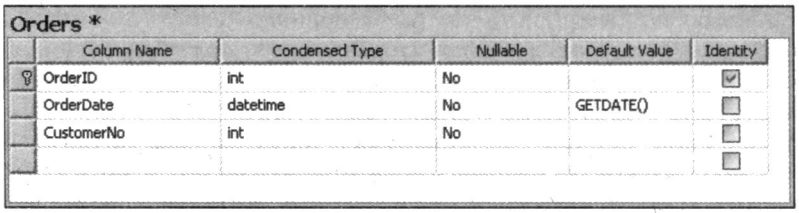


Рис. 8.24. Определение таблицы `Orders`

Следует отметить, что при подготовке данного примера вместо представления, предусмотренного по умолчанию, которое не допускает возможность задавать применяемое по умолчанию значение (столбец `Default Value`) и вводить в действие столбец идентификации (`Identity`), был выполнен переход к специализированному представлению. Автору также пришлось перейти к диалоговому окну Modify Custom (Модифицировать специализированное представление) и выбрать вариант с добавлением столбцов `Default Value` и `Identity` к специализированному представлению.

Мы должны уделить достаточно внимания анализу принятых решений. Безусловно, проблема нормализации была решена, но какие-либо другие важные вопросы еще не выяснены. Прежде всего необходимо решить вопрос об использовании типов данных.

В рассматриваемом случае столбец `OrderID` применяется в качестве первичного ключа таблицы, поэтому необходимо обеспечить, чтобы во внутреннем представлении значений этого столбца было предусмотрено достаточно места для поддержки уникальности значений по мере вставки все большего и большего количества данных. Если бы таблица `Orders` не была предназначена для вставки весьма значительного количества строк, то можно было бы выбрать тип данных с меньшим форматом представления, но эта таблица предназначена для вставки строк с данными о заказах (и можно рассчитывать на то, что количество заказов будет велико), поэтому целесообразно выбрать несколько больший формат представления. Кроме того, создается впечатление, что допустимо использование числовых обозначений номеров заказов (но по такому поводу всегда следует консультироваться с заказчиками), причем для автоматического формирования таких номеров может применяться механизм последовательной выработки чисел в форме столбца идентификации. При условии, что понадобится количество номеров заказов, достигающее 2 миллиардов (оставалось бы только позавидовать акционерам такой компании), потребовался бы тип данных `BigInt` с еще большим форматом представления. (Обычно количество используемых идентификационных номеров, в данном случае номеров заказов, далеко не достигает предельного значения, обусловленного применением конкретного типа данных; следует лишь помнить, что для них необходимо оставлять дополнительное пространство, но в достаточно крупной схеме зачастую ошибки в большую сторону при выборе формата представления идентификаторов в базе данных не считаются достаточно серьезными.)

Что касается столбца `OrderDate`, то первым побуждением обычно становится желание применить тип данных `smalldatetime`. При этом обычно напрашиваются такие соображения, что не требуется точность, которую обеспечивает тип данных `datetime`, кроме того, вряд ли в дальнейшем потребуются данные, относящиеся к далекому прошлому (по-видимому, достаточно будет хранить данные всего лишь за несколько лет). Тем не менее в этом случае используется тип данных `datetime`, а не `smalldatetime`. Это решения принято для облегчения задачи программистов, работающих на языке Visual Basic. В версиях Visual Basic, предшествующих тем версиям, которые вошли в состав инфраструктуры .NET, возникали трудности при использовании данных типа `smalldatetime`. Безусловно, такие сложности были преодолимыми, но для этого требовались определенные трудозатраты. В данном случае столбец `datetime` применяется исключительно ради того, чтобы можно было упростить разработку прикладного программного обеспечения.

Заказчик данного приложения сообщил, что для представления данных в столбце `CustomerNo` достаточно пяти цифровых знаков (и это подтверждается представленными образцами данных). В этом случае не мешало бы также задать вопрос заказчику по поводу того, действительно ли в указанном столбце никогда не будут использоваться буквенные символы. Приняв предположение, что эти значения всегда будут фактически оставаться числовыми, можно воспользоваться для объявления столбца целочисленным типом данных, поскольку он предоставляет описанные ниже преимущества.

- Поиск данных ускоряется.
- Формат внутреннего представления становится короче, поскольку 4 байтов вполне достаточно для представления десятичного числа. состоящего из 5 цифр, а для представления строки длиной 5 символов требуется не меньше 5 байтов (или 6 байтов, если используются символьные данные переменной длины).

Следует отметить, что, рассуждая в данном случае о формате представления, мы рассматриваем лишь пример, поскольку в действительности формат представления номеров заказчиков для этой таблицы уже определен в результате того, что между таблицами `Orders` и `Customers` установлена связь. Но поскольку при описании таблицы `Customers` речь не шла о формате представления номеров заказчиков, это упущение было устранено в настоящем разделе.

Иногда после определения типов данных возникает также необходимость задать размеры столбцов, но для данной конкретной таблицы в этом нет необходимости, поскольку для всех типов данных выбраны постоянные размеры.

На следующем этапе необходимо определить, допустимо ли применение в строке NULL-значений. В рассматриваемом случае можно быть уверенным в том, что в каждой строке обязательной является вся информация и эта информация должна быть доступной ко времени ввода заказа, поэтому использование NULL-значений не допускается.

Безусловно, под этим не подразумевается, что следует отказаться от применения NULL-значений во всех базах данных. В определенных ситуациях практически невозможно обойтись без их использования, поэтому такие "неопределенные" значения являются вполне допустимыми. Тем не менее автор часто предпочитает вводить в таких случаях в текстовые столбцы действительные текстовые значения, которые содержат строки, говорящие о том, что значение не определено (например, "`Value Unknown`" или что-то в этом роде).

Причина, по которой я предпочитаю такой подход, состоит в том, что применение NULL-значений способствует возникновению ошибок в такой же степени, как и применение необъявленных переменных. Обнаружив в таблице неопределенное значение, невозможно определить, было ли это значение введено намеренно или осталось таким, каким было задано по умолчанию, поскольку программист или пользователь просто забыл ввести в таблицу значение для соответствующей строки. В последнем случае имеет место ошибка в программе или упущение в процессе ввода данных.

На следующем этапе должно быть принято решение об использовании заданных по умолчанию значений. Для столбца `OrderID` применяемое по умолчанию значение не может быть предусмотрено, поскольку значение в нем формируется по такому принципу, как в столбце идентификации (а эти два способа автоматического заполнения столбца являются взаимоисключающими). Но для столбца `OrderDate` использование заданного по умолчанию значения имеет смысл. Если при вводе данных о заказе значение `OrderDate` не будет задано, то может быть принято предположение, что датой ввода заказа должна считаться текущая дата. Наконец, что касается столбца `CustomerNo`, то заранее нельзя узнать, от какого заказчика будет принят заказ, поэтому для него применяемое по умолчанию значение не предусмотрено.

После этого перейдем к решению вопроса об использовании столбца идентификации. Для применения в качестве столбца идентификации идеально подходит столбец `OrderID`, поскольку представленные в нем значения предназначены исключительно для обеспечения уникальности строк. Применение счетчика, подобного заданному с помощью столбца идентификации, позволяет реализовать удобный, наглядный и единообразный способ обеспечения уникальности строк. Кроме того, отсутствуют какие-либо основания, требующие изменения начального значения и приращения для столбца идентификации, поэтому эти исходные параметры остаются неизменными. Отсчет значений в столбце начинается с единицы и каждый раз увеличивается на единицу.

Теперь перейдем к рассмотрению следующей таблицы, `OrderDetails`, которая приведена на рис. 8.25.

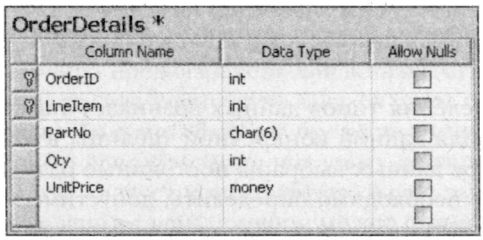


Рис. 8.25. Определение таблицы `OrderDetails`

Столбец `OrderID`, который определен в этой таблице, должен быть обозначен как столбец, на котором объявлен внешний ключ, поэтому решение о выборе для него типа данных уже задано; этот столбец должен иметь такой же тип и размер представления данных, как и столбец, на который он ссылается, иными словами, он должен иметь тип данных `int`.

Отсчет позиций в заказе должен возобновляться с началом ввода каждого нового заказа, поэтому для столбца `LineItem` достаточно предусмотреть применение типа данных `tinyint`. Тем не менее исключительно ради безопасности в этом случае ис-

пользуется тип данных int (поскольку на практике иногда встречаются случаи переполнения из-за превышения величины, допустимой для данных типа tinyint).

Для рассматриваемой таблицы формат представления столбца PartNo фактически уже предопределен в связи с тем, что этот столбец должен согласовываться со столбцом PartNo таблицы Products. В таблице Products для этого столбца будет применяться тип данных char(6) (дополнительная информация об этом приведена ниже), поэтому такой же тип данных выбран и здесь.

При выборе подходящего типа данных для столбца Qty приходится прибегать к рассуждениям. Проблема состоит в том, какое максимальное количество единиц для одного элемента заказа допускается использовать в заказе. В данном случае отсутствует информация о том, какие товары должны быть представлены в заказе, поэтому невозможно определить максимальное количество (например, если с помощью заказа должна быть выполнена поставка нефти, то объем единоразовой поставки может составлять миллионы баррелей). Кроме того, в рассматриваемом случае используется тип данных int, но если бы с помощью заказов проводилась продажа таких товаров, для измерения количества которых применяются десятичные дробные числа (например, единицы объема или веса), то пришлось бы применить более подходящий для этого тип данных.

А что касается столбца UnitPrice, то для него тип данных выбрать несложно, — поскольку в этом столбце должны быть представлены денежные значения, то в качестве типа данных необходимо выбрать money.

Переходя к следующему этапу анализа, снова отметим, что при вводе строк должны быть обязательно заданы значения для всех столбцов (и это неудивительно). Итак, и в данной таблице не допускается использование NULL-значений в какомлибо из столбцов.

Кроме того, очевидно, что для рассматриваемой таблицы также не имеет смысла предусматривать применение заданных по умолчанию значений, поэтому данный этап подготовки определения таблицы пропускается.

Что касается использования столбца идентификации, то может возникнуть стремление обозначить в качестве столбца идентификации также и столбец OrderID этой таблицы. Но такое решение является неприемлемым. Следует помнить, что в столбце OrderID таблицы OrderDetails должны находиться значения, согласующиеся со значениями в одном из столбцов другой таблицы, Orders. В таблице Orders уже заданы значения соответствующего столбца (в настоящем примере он определен как столбец идентификации, но может быть принят и другой способ формирования вводимых в него значений), поэтому в результате определения столбца OrderID таблицы OrderDetails как столбца идентификации возникнет коллизия. Система формирует сообщение об ошибке, свидетельствующее о том, что вставка не может быть выполнена в связи с попыткой применить операцию присваивания к идентификационному значению. А во все другие столбцы данные должны быть перенесены из других таблиц или обязательно введены пользователем. В этом случае значение IsRowGuid снова является неприменимым.

В результате выполнения описанных выше этапов подготовки определений таблиц Products и Customers будут получены таблицы, показанные соответственно на рис. 8.26 и 8.27.

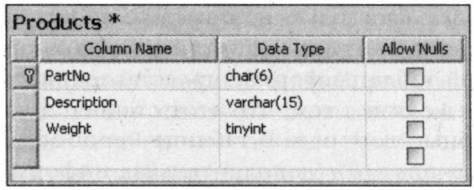


Рис. 8.26. Определение таблицы **Products**

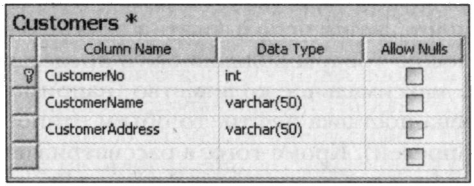


Рис. 8.27. Определение таблицы **Customers**

Рассмотрим кратко, какие соображения послужили причиной выбора типов данных, показанных на рис. 8.26 и 8.27, и после этого перейдем к дальнейшему изложению.

Формат представления значений в столбце `PartNo` (Идентификатор детали) был определен на основании анализа данных, пример которых приведен в разделе с описанием процесса нормализации. Обозначение детали состоит из цифры, за которой следует буква, а за ней стоят четыре цифры. Таким образом, идентификатор детали состоит из шести символов, и есть все основания полагать, что формат этого обозначения является постоянным. Можно было бы добиться от заказчика еще более однозначного подтверждения того, что для идентификации номера детали не потребуется больший формат представления, но примем предположение, что дело действительно обстоит так, и остановимся на типе данных `char(6)`. Такое решение обусловлено тем, что для данных типа `char` требуется немного меньший размер внутреннего представления, чем для `varchar`; к тому же известно, что длина идентификатора всегда остается одинаковой (а это означает, что мы не получим никакого преимущества, выбрав переменный размер).

Данные, предназначенные для хранения в таком столбце, как `Description` (Описание), требуют дополнительного анализа. Иногда выбор формата представления для подобных данных определяется требованиями к пользовательскому интерфейсу (например, зависит от того, какую ширину имеет поле, предназначенное для отображения этих данных на экране), а в других случаях действительно приходится руководствоваться только догадками, касающимися того, является ли выбранный формат представления "достаточным". В данном случае для столбца `Description` используется символьный тип данных переменной длины, а не постоянной длины по следующим двум причинам:

- это позволяет сэкономить немного места;
- не приходится дополнять строку пробелами (дополнительные сведения о типах данных `char` и `varchar` (см. в главе 2).

В рассматриваемом примере типы данных `nchar` и `nvarchar` не применяются, поскольку речь идет о простой системе выставления счетов для предприятия в США, поэтому не приходится представлять символы других национальных алфавитов, для которых требуется кодировка Unicode.

Столбец `Weight` напоминает столбец `Description` в том, что для него приходится также выбирать тип данных, руководствуясь предположениями. В рассматриваемом случае выбран тип данных `tinyint`, поскольку вес товара не должен превышать 255 фунтов. Следует также отметить, что выбор такого типа данных исключает возможность задавать дробные десятичные значения веса (допускаются только целочисленные значения).

Описание требований к столбцу `CustomerNo` было приведено в том разделе, где рассматривалась таблица `Orders`.

При выборе типа данных для столбцов `CustomerName` и `CustomerAddress` приходится сталкиваться почти с такой же ситуацией, как и применительно к столбцу `Description`; проблема состоит в том, что должно быть правильно выбрано максимально допустимое значение длины строк с именем и адресом заказчика. Кроме того, необходимо проследить, чтобы заданное значение длины не оказалось слишком большим.

Как и в других примерах таблиц, значения во всех столбцах должны быть обязательно заданы (не допускается применение NULL-значений ни в одной таблице), а также не предусмотрено использование заданных по умолчанию значений. Кроме того, анализ показывает, что для обозначения идентификатора заказчика и идентификатора детали не допускается применение столбцов идентификации, поскольку указанные идентификаторы имеют специальные форматы и не могут быть сформированы с помощью автоматической системы нумерации, которая поддерживается в столбцах идентификации.

Ввод в действие связей

В целях упрощения рассматриваемых схем автор решил пересмотреть все четыре применяемые таблицы и изменить формат их отображения так, чтобы на экране были показаны только имена столбцов. В программе Management Studio для этого достаточно щелкнуть правой кнопкой мыши на таблице и выбрать во всплывающем меню команду Column Names.

В результате должна быть сформирована диаграмма, которая выглядит так, как показано на рис. 8.28.

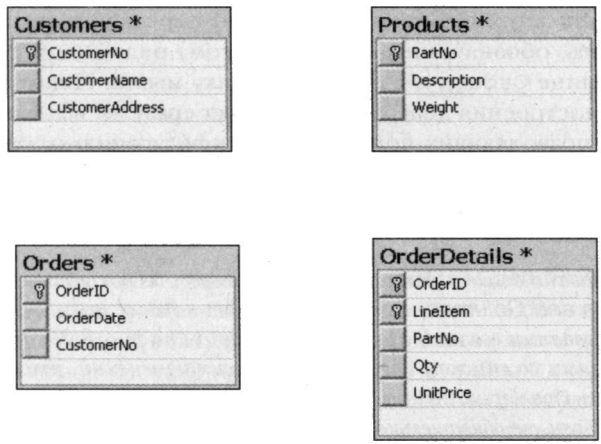


Рис. 8.28. Исходная диаграмма, применяемая в примере создания связей

После того как читатель выполнит указанные действия, позиции. занимаемые таблицами, могут немного отличаться от тех, которые показаны на рис. 8.28, но структура таблиц должна быть точно такой же. Теперь мы можем приступить к решению задачи ввода в действие связей, но, по-видимому, вначале необходимо уделить определенное внимание вопросу о том, какого рода связи действительно требуются.

В данном примере все связи, которые будут отображены с помощью линий связи в окне инструментального средства формирования диаграмм SQL Server, должны быть связями "один к нулю, одному или многим". Фактически в программном обеспечении SQL Server не предусмотрена возможность неявно задавать связи каких-либо других типов. Как уже было описано выше в данной главе, ввод в действие таких объектов, как ограничения уникальности и триггеры, позволяют дополнить те операции, которые реализованы в программном обеспечении SQL Server применительно ко всем связям, но поскольку указанные объекты в данном случае не используются, то, естественно, приходится ограничиваться связями "один к нулю, одному или многим".

Указанное положение дел имеет положительную сторону, поскольку связи "один к нулю, одному или многим" относятся к категории связей наиболее распространенного типа. Иначе говоря, не следует слишком задумываться над тем, что в программном обеспечении SQL Server не поддерживаются связи всех возможных типов. Для обеспечения взаимодействия большинства таблиц вполне достаточно воспользоваться обычным ограничением внешнего ключа (а именно такое ограничение и представляют линии связи на диаграмме). А что касается связей других типов, то их обычно можно промоделировать с помощью других средств.

Начнем с центральной таблицы в рассматриваемой системе выставления счетов — с таблицы `Orders`. Вначале рассмотрим, какие связи могут потребоваться для работы с этой таблицей. В данном случае имеется только одна такая связь, которая позволяет обращаться к таблице `Customers`. Эта связь должна представлять собой связь "один ко многим", в которой таблица `Customers` должна быть родительской (и находиться на стороне "один"), а таблица `Orders` должна быть дочерней (и находиться на стороне "многие").

Для формирования связи (и ограничения внешнего ключа, которое должно служить основанием для этой связи) достаточно нажать и удерживать кнопку мыши на самом левом столбце (в области, обозначенной серым цветом) таблицы `Customers`, т.е. на столбце `CustomerNo`. Затем необходимо перетащить курсор мыши в ту же позицию (в область, обозначенную серым цветом) рядом с обозначением столбца `CustomerNo` в таблице `Orders` и отпустить кнопку мыши. После этого в инструментальном средстве построения диаграмм SQL Server сразу же всплывает первое из двух диалоговых окон, позволяющих подтвердить конфигурацию создаваемой связи. Это первое диалоговое окно, Tables and Columns, показанное на рис. 8.29, дает возможность проверить, действительно ли правильно выбраны столбцы, между которыми устанавливается связь.

Как уже было сказано выше в данной главе, не следует беспокоиться, если имена, показанные в окне Tables and Columns, не соответствуют вашему замыслу, которым вы руководствовались при создании связи. В примере, показанном на рис. 8.29, достаточно просто воспользоваться полями со списком для внесения таких изменений, чтобы на обеих сторонах связи применялись столбцы `CustomerNo`. Следует также отметить, что имена столбцов, участвующих в связи, не обязательно должны быть одинаковыми, но обычно использование одинаковых имен для столбцов двух таблиц, имеющих одинаковое назначение, позволяет избежать путаницы.

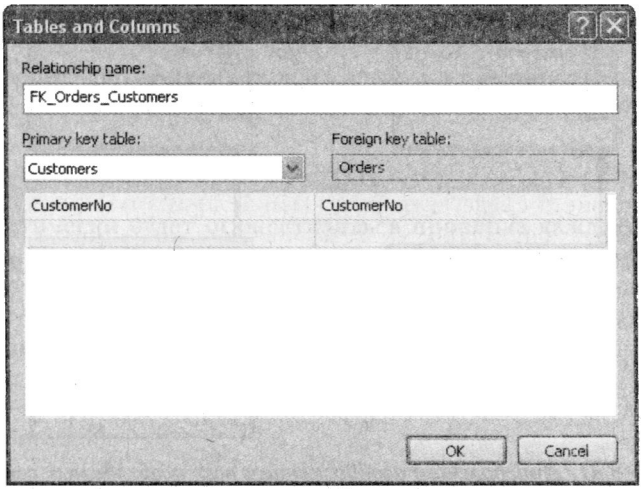


Рис. 8.29. Диалоговое окно Tables and Columns

Щелкните на кнопке OK, чтобы закрыть диалоговое окно, показанное на рис. 8.29. После этого появится диалоговое окно Foreign Key Relationship. Щелкните на кнопке OK также и в этом окне, чтобы подтвердить правильность заданных по умолчанию значений. Как только будет выполнен щелчок на кнопке OK во втором диалоговом окне, во вновь создаваемой базе данных появится первая связь (рис. 8.30).

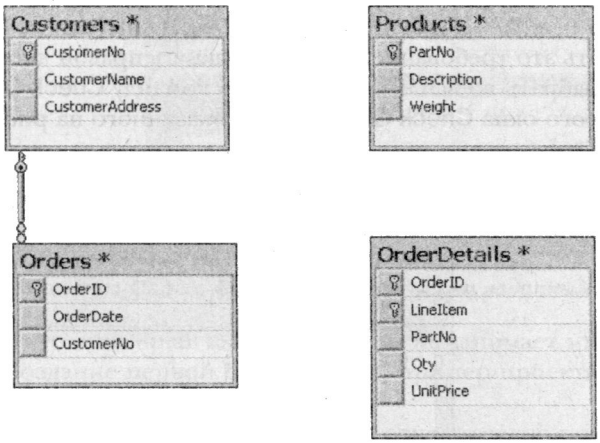


Рис. 8.30. Диаграмма, на которой показана впервые созданная связь

После этого остается только выполнить аналогичные действия для создания двух других связей. Во-первых, необходимо установить связь "один ко многим" от таблицы `Orders` к таблице `OrderDetails` (поскольку под заголовком одного заказа может быть задано много строк расшифровки) на основе столбца `OrderID`. Во-вторых, необходимо сформировать аналогичную связь, проходящую от таблицы `Products` к таблице `OrderDetails` (поскольку одна строка с данными о товаре таблицы `Products` может соответствовать нескольким строкам таблицы `OrderDetails`) на основе столбца `PartNo`, как показано на рис. 8.31.

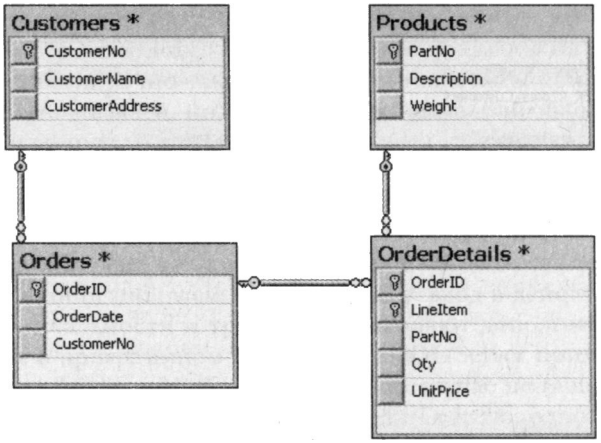


Рис. 8.31. Диаграмма с изображением всех необходимых связей

Ввод в действие некоторых ограничений

В настоящей главе при описании процесса создания таблиц и связей упоминалось одно требование, которое еще не было нами выполнено. Но для реализации этого требования необходимо ввести в действие ограничение. Речь идет о том, какой формат должен быть предусмотрен для представления идентификаторов деталей. Предположим, что идентификаторы деталей имеют формат "9A9999", где "9" обозначает цифру от 0 до 9, а "A" — буквенный (нечисловой) символ.

Чтобы выполнить это требование, следует щелкнуть правой кнопкой мыши на таблице Products и выбрать во всплывающем меню команду Check Constraints для отображения диалогового окна Check Constraints, показанного на рис. 8.32.

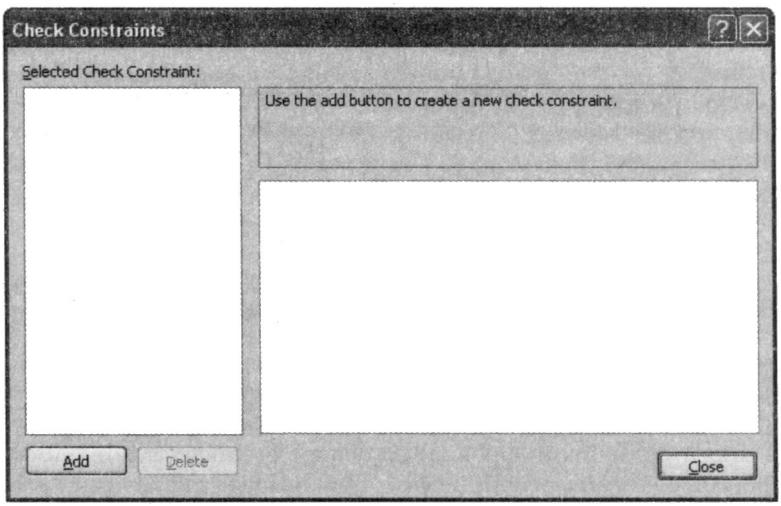


Рис. 8.32. Диалоговое окно Check Constraints

Начиная с этого момента появляется возможность щелкнуть на кнопке Add и ввести в действие рассматриваемое ограничение. Для того чтобы ограничить возможность ввода идентификаторов деталей только такими строками, которые имеют заданный формат, необходимо воспользоваться оператором LIKE, например, как показано ниже.

```
(PartNo LIKE '[0-9][A-Z][0-9][0-9][0-9][0-9]')
```

При этом фактически будет осуществляться проверка каждого символа, который пользователь попытается ввести в столбец PartNo рассматриваемой таблицы. Первый символ должен представлять собой цифру от 0 до 9, второй — букву от A до Z (символ алфавита), а следующие четыре символа снова должны быть представлены цифровыми знаками (от 0 до 9). Для этого достаточно ввести указанную выше строку в текстовое поле, обозначенное как Expression. Кроме того, было решено сменить предусмотренное по умолчанию имя для ограничения с CK_Products на CK_PartNo, как показано на рис. 8.33.

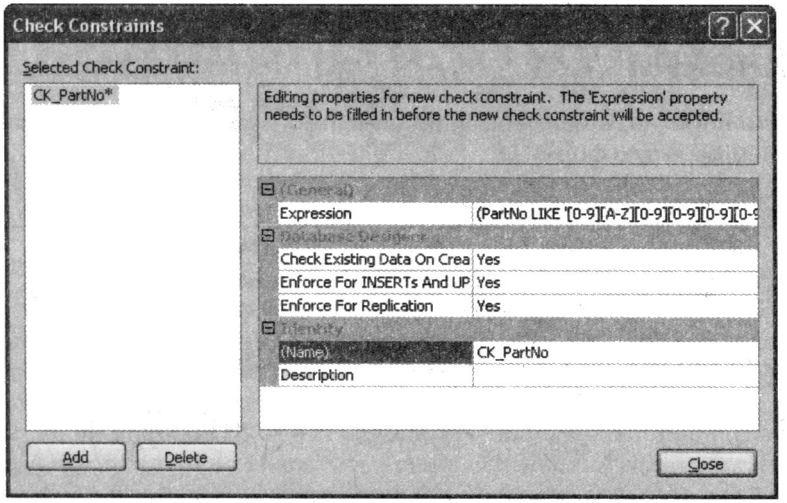


Рис. 8.33. Диалоговое окно **Check Constraints** *со всеми заданными параметрами*

Итак, очевидно, что описанная выше процедура не занимает много времени, и на этом завершается создание первой базы данных, пример проектирования которой рассматривается в настоящей книге.

Безусловно, в данном случае рассматривалась относительно простая модель, но в ходе ее описания применялись операции, которые, скорее всего, позволяют реализовать не менее чем на 90% любую реальную архитектуру данных.

Резюме

Задача проектирования баз данных является чрезвычайно сложной, и одной лишь этой теме посвящено много превосходных книг. Разъяснить каждое понятие, относящееся к проектированию баз данных, всего лишь в одной или двух главах практически невозможно.

Тем не менее настоящая глава позволяет читателю получить солидную начальную подготовку. Прежде всего, в этой главе показано, что данные рассматриваются как нормализованные после их приведения к третьей нормальной форме. На таком уровне обеспечивается устранение дублирующейся информации и данные становятся полностью зависимыми от используемых ключей. Иными словами, нормализованными считаются данные, которые зависят от "ключей, только ключей и ничего другого, кроме ключей". В этой главе было также показано, что нормализацию не всегда можно считать единственным применимым подходом к усовершенствованию базы данных, поскольку тщательно продуманная денормализация данных позволяет упростить работу пользователей с базой данных и повысить производительность решения задач формирования отчетов. Наконец, в этой главе рассматривались некоторые понятия, не связанные с нормализацией, на примере разработки проекта базы данных, а также было указано, как использовать инструментальные средства daVinci для проектирования базы данных.

В следующей главе приведено очень подробное описание того, как обеспечивается хранение информации в СУБД SQL Server и как лучше всего использовать индексы.

Упражнения

8.1. Провести нормализацию данных, приведенных в табл. 8.13, с преобразованием в третью нормальную форму.

Таблица 8.13. Исходные данные для упр. 8.1

Patient	SSN	Physician	Hospital	Treatment	AdmitDate	ReleaseDate
Sam Spade	555-55-5555	Albert Schweitzer	Mayo Clinic	Lobotomy	10/01/2005	11/07/2005
Sally Nally	333-33-3333	Albert Schweitzer	NULL	Cortizone Injection	10/10/2005	10/10/2005
Peter Piper	222-22-2222	Mo Betta	Mustard Clinic	Pickle Extraction	11/07/2005	11/07/2005
Nicki Doohickey	123-45-6789	Sheeze Sheila	Mustard Clinic	Cortizone Injection	11/07/2005	11/07/2005

9

Структуры памяти и индексные структуры SQL Server

Индексы представляют собой исключительно важный компонент средств планирования работы базы данных и сопровождения системы. Индексы позволяют реализовывать дополнительные способы поиска данных и создания кратких обозначений для участков физического расположения данных в SQL Server (а также в определенном смысле, во всех других системах баз данных). Ввод в действие подходящего индекса позволяет существенно сократить время выполнения запросов. Но, к сожалению, на практике при использовании слишком большого количества неправильно запланированных индексов время выполнения запросов может увеличиваться. В действительности индексы, как правило, относятся к той категории объектов СУБД SQL Server, которые многими разработчиками трактуются неправильно и поэтому чаще всего также неправильно применяются.

В настоящей главе приведено довольно подробное описание индексов с точки зрения разработчика и администратора, но для того, чтобы лучше понять, как действуют индексы, необходимо разобраться в том, как организовано хранение данных в СУБД SQL Server. По этой причине в этой главе кратко рассматривается механизм хранения данных СУБД SQL Server.

Средства хранения данных СУБД SQL Server

Можно считать, что данные в СУБД SQL Server представлены с помощью своеобразной иерархии структур, хотя эта иерархия довольно проста. С некоторыми объектами в иерархии структур данных приходится сталкиваться непосредственно,

поэтому такие объекты широко известны. С другой стороны, некоторые другие объекты недоступны для постороннего взгляда, и, несмотря на то, что в определенных случаях к ним может быть предоставлен непосредственный доступ, обычно они остаются недостижимыми. Ниже приведено последовательное описание объектов иерархии структур данных.

База данных

База данных как объект иерархии структур данных является наиболее известной. Автору даже приходилось слышать заявление некоторых специалистов, что они знают о базах данных все. Возможно, они действительно правы, но нельзя забывать, что каждая база данных представляет собой уникальную сущность, поскольку она находится на самом высоком уровне структур хранения данных (на каком-то конкретном сервере). Уровень базы данных — это наивысший уровень, на котором могут устанавливаться **блокировки**, хотя и невозможно явно создать блокировку уровня базы данных.

Блокировка — это объект, одновременно напоминающий замок и маркер места хранения данных, которые используются системой. Специалисты, занимающиеся разработкой программного обеспечения с применением СУБД SQL Server (или, с этой точки зрения, с помощью любой другой СУБД), не сомневаются в том, что для успешной эксплуатации системы абсолютно необходимо знать все особенности блокировок и правильно ими управлять.

Большой объем сведений о блокировках будет приведен в главе 14, а в данной главе возможности задания блокировок на объектах в СУБД SQL Server будут кратко рассматриваться при обсуждении структур памяти.

Экстент

Экстент — это основная единица памяти, используемая при распределении пространства для таблиц и индексов. Экстент состоит из восьми смежных **страниц данных** объемом 64 Кбайт.

Для тех специалистов, которые знакомы только с принципами организации памяти в операционной системе, принятый в СУБД подход к распределению пространства на основе экстентов вместо использования самого пространства может оказаться довольно сложным для понимания. Ниже приведены наиболее важные сведения, касающиеся применения экстентов.

- Если вслед за заполнением очередного экстента возникает необходимость записать следующую строку, то для этой строки выделяется пространство, объем которого определяется не размером самой строки, а размером целого нового экстента. Многие разработчики, впервые приступая к эксплуатации СУБД SQL Server, отчасти ошибаются в своих оценках потребности в пространстве, поскольку не учитывают, что выделение памяти происходит каждый раз в объеме одного экстента, а не одной строки.
- После записи новой строки во вновь выделенный экстент остальной объем памяти остается свободным. Таким образом, происходит заблаговременное распределение пространства, поэтому в СУБД SQL Server достигается экономия времени, поскольку не требуется выполнять операцию распределения пространства памяти после поступления каждой новой строки.

На первый взгляд может показаться, что выделение целого экстента для размещения всего лишь одной строки, в расчете на то, что в дальнейшем в экстент (экстенты), распределенный к настоящему времени, будет добавлено требуемое количество строк, представляет собой непроизводительный способ использования пространства, но обычно объем пространства, который применяется непроизводительно (остается незанятым), не так уж велик. Тем не менее о том, что во многих экстентах может оставаться неиспользуемое пространство, всегда следует помнить, особенно если условия эксплуатации базы данных таковы, что в результате удаления и добавления строк фрагментация пространства в экстентах постоянно увеличивается.

В целом одним из важных преимуществ распределения пространства памяти с помощью экстентов является то, что часть пространства распределяется заблаговременно, поэтому в СУБД SQL Server удается исключить определенную часть затрат времени при выполнении операций распределения памяти. Дело в том, что в СУБД SQL Server операция распределения дополнительного пространства выполняется только после того, как потребуется новый экстент, поэтому не приходится заниматься проблемами распределения пространства при записи каждой новой строки.

Необходимо различать объем пространства, занимаемый экстентами, и объем пространства, который занимает сама база данных. Пространство памяти, выделенное для базы данных на жестком диске, становится недоступным для распределения памяти под какие-либо другие объекты, т.е. объем свободного пространства на жестком диске уменьшается на эту величину. А экстенты, в свою очередь, распределяются в пределах общего объема пространства, зарезервированного для базы данных.

Страница

Во многом аналогично тому, что экстент является единицей распределения памяти в базе данных, страница – это единица распределения памяти в каждом конкретном экстенте. Каждый экстент состоит из восьми страниц.

Страница – это последний уровень иерархии структур памяти, достигаемый перед переходом к строке с фактическими данными. Безусловно, количество страниц в расчете на один экстент является постоянным, но количество строк в расчете на одну страницу изменяется, поскольку полностью зависит от размера строки, который может изменяться. Страницу можно рассматривать как своего рода контейнер для данных строк таблицы и индекса. Но разбивка строк, в результате которой ее данные переходили бы с одной страницы на другую, не допускается.

Страницы подразделяются на несколько типов. В настоящей книге рассматриваются только типы страниц, описанные ниже.

- **Страницы данных** в основном не требуют пояснений. Фактически страницы данных обеспечивают хранение данных, относящихся к таблице. К данным, размещаемым на страницах, не относятся только данные типа BLOB, которые не определены с помощью опции "`text in row`" (текст, заданный в строке) или параметра `varchar(max)`.
- **Страницы индекса** также в основном не требуют пояснений. Они применяются для хранения страниц листового и нелистового уровней некластеризованного индекса, а также страниц нелистового уровня кластеризованного индекса (информация по этой теме будет приведена ниже в данной главе). Особенности индексов этих типов будут становиться все более понятными по мере дальнейшего изучения материала, приведенного в настоящей главе.

Разбиение одной страницы на две

После заполнения страницы происходит ее разбиение на две страницы. Операция разбиения страницы не сводится только к тому, что распределяется новая страница; в результате разбиения примерно половина данных переписывается с существующей страницы на новую.

Но если применяется кластеризованный индекс, процесс разбиения страниц происходит иначе. Если на таблице задан кластеризованный индекс и возникает такая ситуация, что очередная вставляемая строка должна быть физически расположена в качестве последней строки в таблице, то распределяется еще одна страница и новая строка записывается на эту страницу, причем перемещение каких-либо существующих данных не происходит. Дополнительные сведения о разбиении страниц будут приведены при описании индексов.

Строки

Специалисты, разрабатывающие программное обеспечение для баз данных, часто сталкиваются с таким понятием, как блокировка уровня строк, поэтому знают, что строка представлена в базе данных как отдельный объект. Строки могут иметь размеры до 8 Кбайт.

Более точно указанный выше предел определяется как равный 8060 байтам; кроме того, максимальное количество столбцов составляет 1024. Но на практике чаще приходится сталкиваться с такой ситуацией, что достигается предел длины строки, составляющей 8060 байтов, чем предельное количество столбцов. При количестве столбцов, равном 1024, средняя ширина столбца составляет 8 байтов. Но нелегко найти пример приложения, в котором это значение размера, применяемое для представления данных, было бы достаточным. Обычно исключением из этого правила являются результаты измерений и статистическая информация, поскольку такие приложения требуют, чтобы в базе данных можно было представить большое количество различных показателей для хранения применяемых числовых значений. Но даже в таких приложениях предел количества столбцов, равный 1024 столбцам, достигается очень редко.

Общие сведения об индексах

В толковом словаре Вебстера индекс определяется следующим образом.

Индекс — это список (такой как библиографическая информация или перечень цитат, взятых из литературы), который обычно располагается в алфавитном порядке перечисления определенных данных (таких как авторы, темы или ключевые слова).

Автор обычно использует более простое толкование понятия индекса, в котором применяются термины, относящиеся к базе данных, и определяет индекс как средство, позволяющее ускорить доступ к данным. Тем не менее трактовка понятия индекса, приведенная в словаре Вебстера, не так уж плоха, даже с точки зрения ее применимости в данном контексте.

По-видимому, наибольшего внимания в определении, имеющемся в словаре Вебстера. заслуживает то, что в нем есть слово "обычно". А трактовка понятия "алфавитный порядок" изменяется в зависимости от того, какие конкретные правила применяются для определения самого алфавитного порядка. Например, в СУБД SQL

Server предусмотрена возможность использовать целый ряд различных опций упорядочения данных. Некоторые из этих опций описаны ниже.

- Порядок, учитывающий двоичные значения. При использовании такого подхода сортировка данных происходит с учетом внутреннего представления символа (например, в кодировке ASCII пробел представлен числом 32, буква "D" — числом 68, буква "d" — числом 100). А поскольку компьютеры в наибольшей степени приспособлены для обработки числовых данных, то эта опция обеспечивает также наибольшее быстродействие. Но, к сожалению, формат внутреннего представления не всегда соответствует приемлемой последовательности сортировки данных; кроме того, применение такой опции при проведении операций сравнения в конструкции WHERE действительно может привести к путанице.
- Порядок, в котором учитывается лексикографическое определение. При использовании этой опции сортировка данных происходит по такому же принципу, как и в словаре, но с учетом некоторых дополнительных возможностей. В частности, предусмотрена возможность задавать различные дополнительные опции, позволяющие определить, учитываются ли регистр, диакритический знак и набор символов.

Вполне очевидно, что после передачи СУБД SQL Server указания, что должен учитываться регистр, буква "A" будет рассматриваться как отличная от "a". Аналогичным образом, если указано, что регистр не учитывается, то буква "A" рассматривается как равная букве "a". Задача определения правильного порядка сортировки немного усложняется после ввода в действие средств учета диакритических знаков, иными словами, после указания на то, что в СУБД SQL Server должны учитываться диакритические знаки и поэтому буква "a" рассматривается как отличная от "á", которая отличается от "à". Для многих пользователей еще более значительным источником путаницы является то, что от принятого способа упорядочения зависит не только трактовка понятия равенства символов данных, но и применяемый порядок сортировки (и, тем самым, принятый способ хранения данных в индексах).

В качестве примера рассмотрим, какие правила определяют равенство символов при использовании нескольких различных опций упорядочения, приведенных в табл. 9.1. Кроме того, в этой таблице приведена информация о порядке сортировки и о том, какие условия равенства применяются при выполнении операций сравнения данных.

Таблица 9.1. Правила определения равенства символов при использовании различных опций упорядочения

Схема упорядочения	Сравниваемые значения	Последовательность хранения в индексе
Лексикографический порядок, без учета регистра, без учета диакритического знака (значение, применяемое по умолчанию)	A=a=à=á=â=A=a=Å=å	a, A, à, â, á, A, a, Å, å
Лексикографический порядок, без учета диакритического знака, первоочередное расположение символов верхнего регистра	A=a=à=á=â=A=a=Å=å	A, a, à, â, á, A, a, Å, å
Лексикографический порядок, с учетом регистра	A_a, A_a, Å_å, a_à_á_â_a_å, A_A_Å	A, a, à, á, â, Ä, a, Å, å

Итак, в соответствии с приведенными выше сведениями необходимо учитывать то, как влияет на использование индексов та информация об упорядочении, которая определена для данных, представленных в базе данных. Опции упорядочения могут быть определены на уровне базы данных и на уровне столбца, поэтому разработчик имеет в своем распоряжении довольно мощные возможности по определению степени детализации управления опциями упорядочения. Но если будет решено задать опцию, предусматривающую отсутствие чувствительности к регистру на уровне сервера, то об этом следует обязательно упомянуть в документации на разрабатываемую систему. Если это требование не будет учтено, то необходимо будет подготовиться к приему бесчисленных звонков в службу технической поддержки, особенно если продажа программного продукта осуществляется за пределами того государства, в котором оно разработано. Чтобы понять важность этого требования, достаточно представить себе, что какой-либо независимый поставщик программного обеспечения (Independent Software Vendor — ISV) выполнил поставку вашего программного продукта заказчику, который установил этот продукт на существующем сервере (а это вполне разумное решение, поскольку вряд ли кто-либо станет покупать отдельный компьютер для каждой программы), но на этом действующем сервере применяются опции, предусматривающие учет регистра. В таком случае вам придется приготовиться к тому, что будут поступать жалобы от очень недовольных заказчиков.

После того как база данных, в которой задан определенный способ упорядочения, примет некоторый объем данных, задача перехода к другому способу упорядочения становится весьма нетривиальной (хотя и разрешимой), поэтому необходимо тщательно обосновать применяемый способ упорядочения и только после этого определить соответствующую опцию.

B-деревья

Очевидно, что подход к созданию индексов, основанный на использовании сбалансированного дерева (Balanced Tree — B-Tree), впервые был применен задолго до создания программного обеспечения SQL Server. А в настоящее время сбалансированные деревья (или сокращенно B-деревья) используются в очень многих системах индексации, относящихся и не относящихся к миру баз данных.

Вообще говоря, B-дерево — это структура данных, позволяющая использовать единообразные и относительно недорогие методы поиска способов выборки конкретных фрагментов информации. Само то, что в названии этой структуры данных применяется слово "сбалансированный" (Balanced — B), говорит о многом. Дело в том, что такая структура данных, как B-дерево, обеспечивает автоматическую поддержку сбалансированности узлов. Это означает, что от каждого узла всегда исходит примерно одинаковое количество ветвей; иными словами, примерно половина данных находится по одну сторону от узла и примерно половина — по другую сторону. Кроме того, визуальное представление такой сбалансированной структуры данных также весьма напоминает дерево (состоящее из ветвей и узлов). И действительно, после того, как структура данных B-дерева, изображенная в том виде, в каком ее принято вычерчивать (с ветвями, направленными вниз), будет перевернута, она принимает общую форму дерева.

Формирование B-дерева начинается с образования **корневого узла** (в этом заключается еще одна аналогия с деревом, растущим из одного корня, но далеко не послед-

няя). Если количество данных, доступ к которым осуществляется с помощью индекса, невелико, то найти непосредственное указание на действительное местонахождение искомых данных можно сразу же с помощью корневого узла. В таком случае в конечном итоге формируется структура, которая выглядит так, как показано на рис. 9.1.

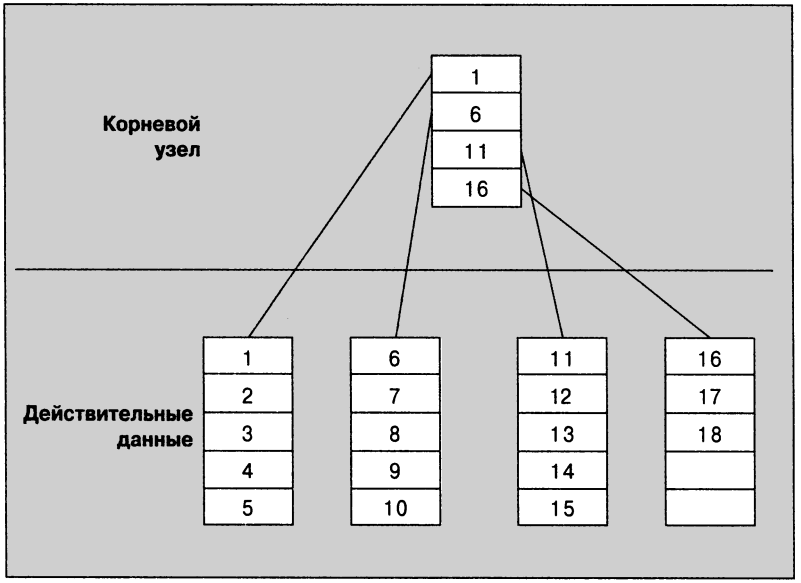


Рис. 9.1. Структура индекса, состоящего из одного корневого узла

Итак, поиск в индексе начинается с корневого узла, после чего осуществляется просмотр строк в этом узле до тех пор, пока не обнаруживается указатель на последнюю страницу, которая начинается со значения меньше искомого. Затем происходит выборка указателя на узел, содержащий сведения об этой странице, и просмотр ее до тех пор, пока не обнаружится требуемая строка.

Но в большинстве ситуаций количество данных в столбце, поиск в котором осуществляется с помощью индексов, слишком велико для того, чтобы можно было поместить все указатели на эти данные в корневой узел, поэтому корневой узел применяется для хранения указателей на промежуточные узлы, или так называемые **узлы уровня, отличного от листового**. Узлы уровня, отличного от листового, представляют собой узлы, находящиеся на одном из уровней между корнем и тем узлом, который позволяет узнать, где физически хранятся данные. В свою очередь, узлы уровня, отличного от листового, могут указывать либо на узлы другого уровня, отличного от листового, либо на **узлы листового уровня** (в этом состоит последний обещанный аналог с деревом, поскольку дерево оканчивается листьями). Узлы листового уровня представляют собой узлы, позволяющие получить реальную ссылку на фактические физические данные. Во многом аналогично тому, что процесс перемещения по ветвям дерева оканчивается достижением отдельного листа, процесс продвижения по индексу оканчивается достижением узла листового уровня, а от этого узла индекса мы можем непосредственно перейти к узлу с данными, в котором хранятся действительные искомые данные.

Как показано на рис. 9.2, поиск данных в многоуровневом индексе, как и прежде, начинается с корневого узла, после этого происходит переход к узлу, который начинается с наивысшего значения, меньшего или равного искомому значению и, кроме того, находящемуся на следующем, нижележащем уровне. После этого данный процесс повторяется: отыскивается узел, имеющий наибольшее начальное значение, меньшее или равное искомому значению. Этот процесс перемещения с одного уровня дерева на другой, нижележащий уровень продолжается вплоть до листового уровня; на этом уровне обнаруживается физическое местонахождение данных, после чего может быть быстро выполнен переход к этим данным.

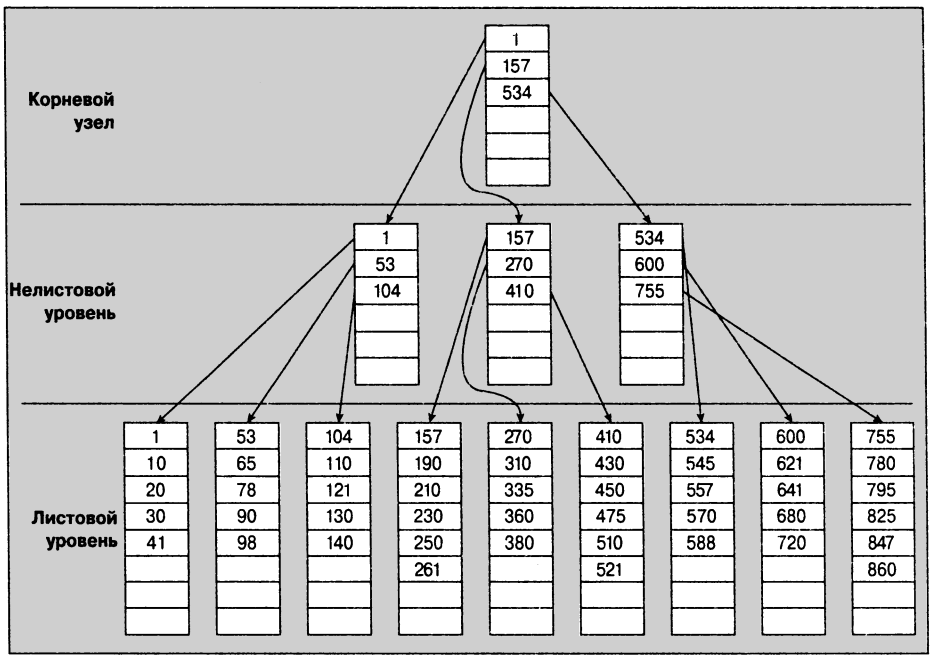


Рис. 9.2. Поиск данных в многоуровневом индексе

Общее описание процесса разбиения страницы

Очевидно, что описанный процесс осуществляется столь бесперебойно, когда происходит чтение данных с помощью индекса, но операция обновления индекса является немного более сложной. Напомним, что буква B в слове B-дерево означает balanced (сбалансированный). Кроме того, как уже было сказано, B-дерево является сбалансированным, поскольку примерно половина данных находится по одну или по другую сторону от каждого узла дерева, из которого исходят ветви к узлам нижележащего уровня. Следует отметить, что B-деревья иногда называют **самобалансирующимися**, поскольку для вставки новых данных в дерево применяется такой метод, который обычно предотвращает появление асимметрии в дереве.

По мере ввода в дерево новых данных происходит заполнение узлов, в результате чего в узлах в конечном итоге больше не остается свободного места и требуется их разбиение. Поскольку в СУБД SQL Server узел индекса представлен в виде страницы,

то вместо указания на то, что происходит разбиение узла, принято применять термин **разбиение страницы**. Процесс разбиения страницы показан на рис. 9.3.

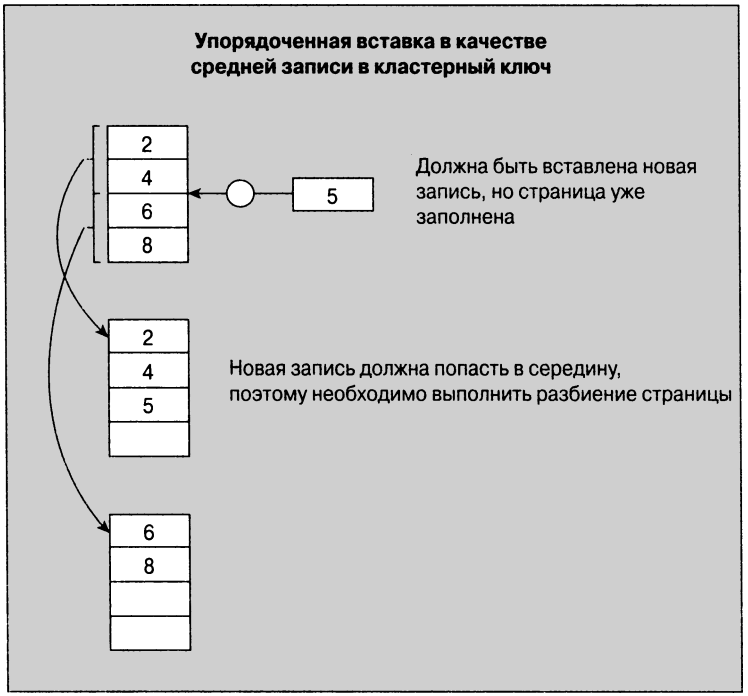


Рис. 9.3. Процесс разбиения страницы

При каждом разбиении страницы автоматически происходит перемещение данных для обеспечения поддержки сбалансированности дерева. Первая половина данных остается на старой странице, а остальные данные переносятся на новую страницу. Таким образом, разбиение осуществляется примерно наполовину и дерево остается сбалансированным.

Даже краткие размышления по поводу того, как происходит процесс разбиения, позволяют понять, что во время разбиения возникают значительные издержки. Дело в том, что в процессе разбиения не происходит лишь вставка одной страницы, а выполняются следующие действия:

- создание новой страницы;
- перенос части строк с существующей страницы на новую страницу;
- добавление новой строки к одной из страниц;
- добавление еще одной строки в родительский узел.

Но на этом работа не заканчивается. Поскольку узлы образуют древовидную структуру, создается возможность своего рода каскадных действий. После того как создается новая страница (в результате разбиения), возникает необходимость ввести еще одну строку в родительский узел. Появление новой строки в родительском узле также может привести к возникновению необходимости осуществить разбиение страницы, но уже на уровне родительских узлов, поэтому процесс разбиения возобновляется на

более высоком уровне. В действительности такая последовательность операций может продолжаться, охватывая все более и более высокие уровни, и в конечном итоге может даже затронуть корневой узел.

Если происходит разбиение корневого узла, то фактически в связи с этим создаются две дополнительные страницы. Но поскольку количество корневых узлов не может быть больше единицы, то страница, на которой перед этим находились строки корневого узла, разбивается на две страницы, а уровень, на котором она находится, становится новым промежуточным уровнем дерева. После этого создается совершенно новый корневой узел, в котором содержатся две строки (одна из них относится к старому корневому узлу, а вторая – к странице, полученной в результате разбиения).

Вполне очевидно, что выполняемые каскадно операции разбиения страниц могут оказывать весьма негативное влияние на производительность системы. Характерным признаком того, что происходит каскадное разбиение, является ситуация, в которой создается впечатление, что процессы обработки данных на сервере кажутся приостановившимися на несколько секунд (на то время, когда происходит разбиение и перезапись страниц).

Информация о том, как предотвратить возникновение ситуаций разбиения страниц, приведена в конце данной главы.

Безусловно, операции разбиения страниц на листовом уровне происходят очень часто, но такие операции в узлах промежуточных уровней происходят гораздо реже. По мере увеличения размеров таблицы операции разбиения страниц осуществляются на каждом уровне индекса, но в узлах промежуточных уровней имеется только по одной строке в расчете на несколько строк узла более низкого уровня, поэтому количество страниц, подвергаемых разбиению, становится все меньше и меньше по мере дальнейшего продвижения вверх по дереву. К тому же разбиение на уровне выше листового может иметь место лишь в том случае, если произошло разбиение на более низком уровне. Это означает, что операции разбиения страниц, осуществляемые на более высоких уровнях дерева, по своему характеру являются кумулятивными, т.е. возникающими в результате осуществления ряда подобных операций (и поэтому их выполнение связано с определенным снижением производительности).

В СУБД SQL Server предусмотрен целый ряд различных типов индексов (информация об этом будет вскоре приведена), но в индексах всех этих типов в той или иной форме используется подход, основанный на применении B-деревьев. На самом деле все эти индексы весьма напоминают друг друга по своей структуре и вместе с тем обладают заметными отличительными особенностями, поскольку B-деревья позволяют реализовывать исключительно разнообразные требования. Однако, как будет описано ниже, некоторые различия между индексами разных типов действительно являются весьма существенными. Кроме того, от типа применяемого индекса во многом зависит производительность системы.

Следует отметить, что в индексах SQL Server узлы дерева представлены в форме страниц, а сами индексы имеют древовидную структуру, которая включает корневой узел, узлы уровней, отличных от листового, и узлы листового уровня. Но подобный принцип организации данных может применяться не только в SQL Server, но и в других базах данных и даже в системах представления данных, отличных от баз данных.

Принципы организации доступа к данным в СУБД SQL Server

Вообще говоря, в СУБД SQL Server применяются только два указанных ниже способа выборки запрашиваемых данных.

- Выборка данных путем полного просмотра таблицы.
- Выборка данных с помощью индекса.

Метод выборки данных, применяемый в СУБД SQL Server для выполнения конкретного запроса, зависит от состава доступных индексов, от того, в каких столбцах находятся требуемые данные, какого рода соединения выполняются и какие размеры имеют таблицы.

Использование полного просмотра таблицы

Процесс полного просмотра таблицы является довольно несложным. При осуществлении полного просмотра таблицы СУБД SQL Server начинает свою работу с физического начала таблицы, после чего считывает каждую строку таблицы. После обнаружения строк, соответствующих критериям запроса, СУБД включает эти строки в результирующий набор.

Широко распространено такое мнение, что операции выборки данных с помощью полного просмотра таблицы характеризуются существенными недостатками. Такое мнение соответствует действительности. Тем не менее операции с полным просмотром таблицы могут фактически оказаться в некоторых обстоятельствах самым быстрым методом доступа. Как правило, именно это происходит при выборке данных из относительно небольших таблиц. Точные данные о размерах таблицы, при которых способ выборки данных с полным просмотром становится наиболее быстродействующим, в основном зависят от размеров строк таблицы и от характерных особенностей самого запроса.

Попытайтесь сами определить, почему применение оператора `EXISTS` в конструкции `WHERE` запроса оказывает такое влияние на производительность при переходе от одного способа выборки к другому. Дело в том, что при использовании оператора `EXISTS` поиск данных в СУБД SQL Server прекращается сразу после обнаружения хотя бы одной строки, соответствующей критериям поиска. Если ведется обработка таблицы, состоящей из миллиона строк, и строка, соответствующая критериям, обнаруживается на третьем месте, то использование опции `EXISTS` позволяет исключить необходимость чтения 999 997 строк! Опция `NOT EXISTS` действует в основном по такому же принципу.

Использование индексов

Если планировщик запросов SQL Server принимает решение об использовании индексов, то процесс выборки данных осуществляется фактически во многом аналогично тому, как происходит полный просмотр таблицы, с учетом определенных способов сокращения.

В процессе оптимизации запроса программа-оптимизатор просматривает все доступные индексы и выбирает из них наилучший (при этом в основном используется информация, заданная в операциях соединения и в конструкции `WHERE`, в сочетании со статистической информацией об устройстве индекса, которая ведется в СУБД SQL Server). После выбора применяемого индекса СУБД SQL Server переходит по древовидной структуре к тому узлу индекса, который указывает на данные, соответствую-

щие установленным критериям. При этом снова извлекаются только необходимые строки. Различие между двумя способами выборки данных состоит в том, что машина выполнения запросов определяет момент достижения конца текущего искомого диапазона немного иначе, поскольку данные отсортированы. После этого может быть завершено выполнение запроса или в случае необходимости осуществлен переход к следующему диапазону определения данных.

Возвратившись к тому изложению тематики запросов, которое было приведено выше в данной книге (особенно в главе 7), можно отметить, насколько отчетливо напоминают действия, выполняемые при выборке данных с помощью индекса, сам способ применения опции EXISTS. Если в запросе задано ключевое слово EXISTS, то разрешается прекратить выполнение запроса сразу же, как только обнаруживается строка, соответствующая критериям запроса. Достигаемое благодаря этому повышение производительности при выборке данных с помощью индекса становится аналогичным или даже лучшим, чем при полном просмотре, поскольку процесс поиска данных может осуществляться по такому же принципу. Иными словами, если задано ключевое слово EXISTS, то серверу фактически дается указание, что после обнаружения хотя бы одной строки, соответствующей критериям, остальные строки не представляют больше интереса и поэтому обработка данных может быть сразу же прекращена. Но еще более значительным преимуществом использования индекса является то, что мы можем не ограничиваться ситуациями однозначного принятия решений (в которых требуется лишь определить, существует ли искомый фрагмент данных, "да" или "нет"). Иными словами, тот же подход можно применить для определения начала и конца некоторого диапазона, а возможность осуществлять выборку данных, относящихся к заданным диапазонам, предоставляет такие же преимущества, какие достигаются при использовании индекса для поиска данных. Более того, можно осуществлять очень быстрый поиск данных (такие операции известны под названием SEEK), а не просматривать всю таблицу.

По существу, в предыдущем абзаце было приведено сравнение возможностей индексов и оператора EXISTS, но я не хочу, чтобы у читателя создалось впечатление, будто оператор EXISTS способен полностью заменить индексы (или наоборот). Эти два средства обработки данных не являются взаимоисключающими; индексы и оператор EXISTS могут использоваться совместно и часто так и используются. Речь об этих двух средствах доступа ведется в одном контексте лишь потому, что индексы и оператор EXISTS позволяют принять решение о том, что задание выполнено, и прекратить обработку таблицы до того, как закончится просмотр всех строк таблицы.

Типы индексов и переход по индексам

Формально считается, что в СУБД SQL Server предусмотрены два типа индексов (**кластеризованный** и **некластеризованный**), но фактически индексы SQL Server по своему внутреннему устройству подразделяются на три различных типа, указанных ниже.

- Кластеризованные индексы.
- Некластеризованные индексы, которые подразделяются на следующие типы:
  - некластеризованные индексы, заданные на неупорядоченной таблице;
  - некластеризованные индексы, заданные на кластеризованном индексе.

В кластеризованных и некластеризованных индексах применяются разные способы физического хранения данных. Кроме того, все три типа индексов отличаются друг от друга тем, что в них СУБД SQL Server по-разному осуществляет переход по B-дереву для достижения конечных данных.

Все индексы SQL Server имеют страницы листового уровня и нелистовых уровней. Как уже было сказано при описании B-деревьев, листовым называется уровень, на котором хранится "ключ", идентифицирующий конкретную строку, а страницы нелистовых уровней используются в качестве указателей, направляющих к листовому уровню.

Индексы создаются либо на кластеризованной таблице (таковой называется таблица, имеющая кластеризованный индекс), либо на так называемой неупорядоченной таблице (так называется таблица, не имеющая кластеризованного индекса).

Кластеризованные таблицы

Кластеризованной называется любая таблица, на которой задан кластеризованный индекс. Подробное описание кластеризованных индексов приведено далее в этой главе, но по существу применение кластеризованного индекса влечет за собой то, что данные в таблице, на которой он задан, хранятся физически в указанном порядке. Отдельные строки в таблице однозначно идентифицируются с помощью **кластеризованного ключа**; так называются столбцы, которые определяют кластеризованный индекс.

Изучение приведенных выше данных невольно наводит на мысль, что применение кластеризованного индекса, не обеспечивающего уникальную идентификацию строк, может привести к нарушению в работе. Дело в том, что если кластеризованный индекс не является уникальным, то не может использоваться для однозначной идентификации строки. Решение этой проблемы заключается в том, что в СУБД SQL Server используются некоторые механизмы, скрытые от пользователя. В частности, СУБД SQL Server принудительно поддерживает уникальность каждого кластеризованного индекса, даже если он по определению не является уникальным. К счастью, применяемый при этом способ не влияет на то, как фактически происходит работа с самим индексом. Пользователь по-прежнему может при желании вставлять строки с повторяющимися значениями индекса, а СУБД SQL Server добавляет к ключу внутренний суффикс для обеспечения того, чтобы каждая строка имела уникальный идентификатор.

Неупорядоченные таблицы

Неупорядоченной таблицей называется любая таблица, на которой не задан кластеризованный индекс. При использовании таких таблиц создается уникальный идентификатор, называемый идентификатором строки (Row ID — RID). Идентификатор RID формируется на основе использования данных об экстенте, странице и смещении строки (измеряемом в позициях от начала страницы) для данной строки. Идентификатор RID требуется, только если отсутствует кластеризованный ключ (на таблице не задан кластеризованный индекс).

Кластеризованные индексы

Для каждой конкретной таблицы **кластеризованный индекс** является уникальным; на каждой таблице может быть задан только один кластеризованный индекс. На таблице не обязательно требуется задавать кластеризованный индекс, но практика

показывает, что именно кластеризованный индекс чаще всего выбирают в качестве первого индекса самых разных таблицах по многим причинам, которые станут очевидными после изучения индексов различных типов.

Отличительной особенностью кластеризованного индекса является то, что на его листовом уровне находятся действительные данные. Иначе говоря, перед записью в таблицу данные сортируются в целях обеспечения их хранения в том физическом порядке, который определяется критериями сортировки индекса. Это означает, что после достижения листового уровня индекса поиск заканчивается, поскольку в вашем распоряжении уже находятся требуемые данные. Вставка каждой новой строки в таблицу осуществляется с учетом ее правильного физического расположения в кластеризованном индексе. При этом задача создания новых страниц решается по-разному, с учетом того, где должна быть выполнена вставка строки.

В том случае, если новая строка должна быть вставлена в средней части индексной структуры, происходит обычная операция разбиения страницы. Вторая половина строк перемещается со старой страницы на новую страницу, а новая строка записывается на новую или старую страницу, в зависимости от того, к какой из них она относится.

А в том случае, если новая строка логически должна быть добавлена в конце индексной структуры, создается новая страница, но на ней записывается только новая строка (рис. 9.4).

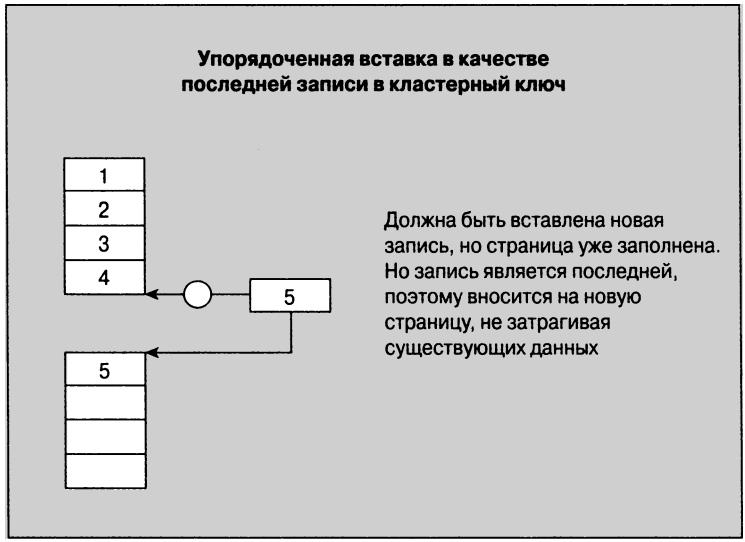


Рис. 9.4. Добавление новой строки в конце индексной структуры

Переход по дереву

Как уже было сказано, в кластеризованных таблицах SQL Server в виде B-дерева хранятся не только индексы, но и данные. Теоретически B-деревья обеспечивают такой способ хранения данных, что под каждой ветвью дерева, исходящей из узла, всегда хранится примерно одинаковый объем информации. На рис. 9.5 приведено графическое представление кластеризованного индекса в виде B-дерева.

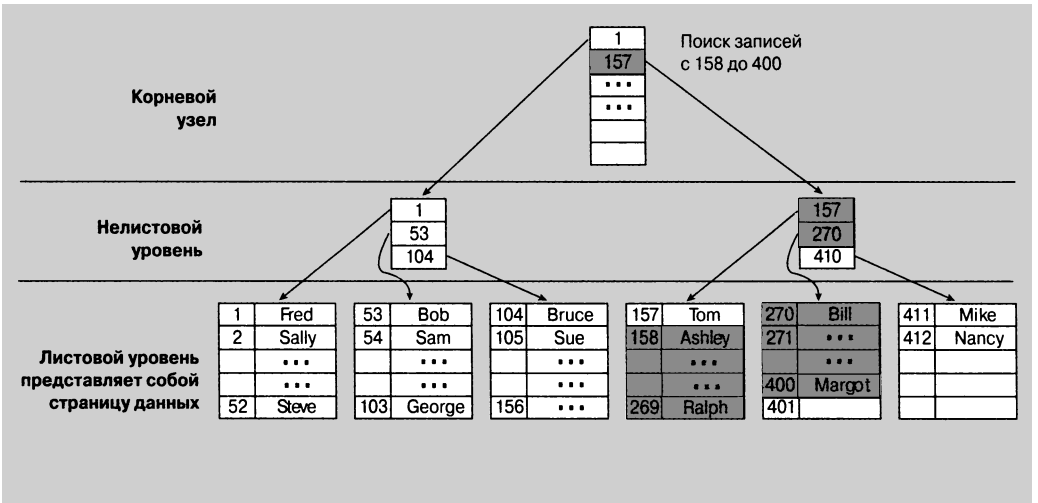


Рис. 9.5. Графическое представление кластеризованного индекса в виде B-дерева

Вполне очевидно, что по своему виду B-дерево кластеризованного индекса фактически является идентичным B-деревьям более общего типа, которые рассматривались выше в данной главе. На рис. 9.5 показано, как выполняется поиск в диапазоне значений ключа от 158 до 400 (кластеризованные индексы являются наиболее подходящими для выполнения этой операции). Для этого достаточно перейти к первой искомой строке, а затем включить в состав обрабатываемых данных все оставшиеся строки на текущей странице. То, что нам потребуется вся оставшаяся часть страницы, можно легко определить, поскольку информация из узла, находящаяся на один уровень выше, позволяет узнать, что потребуются данные из других страниц. Значения ключа в индексе представляют собой упорядоченный список, поэтому можно не сомневаться в том, что эти значения являются последовательными. Таким образом, если на следующей странице имеются строки, которые должны быть включены в область поиска, то тем более должна быть включена оставшаяся часть текущей страницы. Поэтому мы можем приступить к обработке данных, находящихся на страницах, выявленных в процессе поиска, не задумываясь над тем, нужно ли проверять каждую отдельную строку.

Начнем с перехода к корневому узлу. В СУБД SQL Server возможность найти корневой узел обеспечивается благодаря тому, что в системной таблице `sysindexes` хранится информация об этом узле.

> Каждому индексу, применяемому в базе данных, соответствует одна строка в таблице `sysindexes`. Эта системная таблица входит в состав пользовательской базы данных (а не базы данных `master`). В этой таблице хранится информация о местонахождении всех индексов базы данных и о том, на каких столбцах они основаны.

Просматривая страницу, соответствующему корневому узлу, можно выяснить, какая следующая страница требует проверки (как показано на рис. 9.5, таковой является вторая страница на втором уровне). После этого процесс поиска продолжается с найденной страницы. По завершении каждого этапа поиска происходит переход ко

все более низким уровням дерева, которые охватывают все меньшие и меньшие подмножества данных.

Наконец, достигается листовой уровень индекса. В рассматриваемом случае индекс является кластеризованным, поэтому переход на листовой уровень индекса равносилен также достижению требуемой строки (строк) и требуемых данных.

Та особенность кластеризованных индексов, в соответствии с которой полный переход по индексу представляет собой полный переход к искомым данным, является чрезвычайно важной, поскольку способствует значительному повышению производительности. Каковым фактически является это повышение производительности, можно узнать, проведя сравнение с некластеризованными индексами. Разница становится особенно заметной, если некластеризованный индекс создан на основе кластеризованного индекса.

Некластеризованные индексы, заданные на неупорядоченной таблице

Некластеризованные индексы, заданные на неупорядоченной таблице, во многом аналогичны кластеризованным индексам. Тем не менее между индексами этих двух типов наблюдаются некоторые заметные различия, описанные ниже.

Прежде всего, на листовом уровне некластеризованного индекса, заданного на неупорядоченной таблице, не находятся данные. Вместо этого листовой уровень применяется для получения указателя на искомые данные. Указатель представлен в форме идентификатора строки (RID), который, как уже было описано выше в настоящей главе, состоит из данных об экстенте, странице и смещении строки, относящихся к конкретной строке, на которую указывает индекс. Даже несмотря на то, что на листовом уровне отсутствуют действительные данные (вместо этого на этом уровне определены идентификаторы строк), для достижения искомых данных требуется пройти на один шаг больше по сравнению с той ситуацией, когда используется кластеризованный индекс; но поскольку в идентификаторе RID содержится полная информация о местонахождении строки, то обеспечивается возможность перейти непосредственно к данным.

Но из того, что при использовании некластеризованного индекса, заданного на неупорядоченной таблице, приходится, как было сказано выше, осуществлять "еще один шаг", не следует, что при его применении издержки увеличиваются лишь незначительно и что этот индекс обеспечивает почти такое же быстродействие, как кластеризованный индекс. Еще раз отметим, что данные, на которых определен кластеризованный индекс, имеют физическое расположение, соответствующее последовательности ключей в индексе. Это означает, что при выборке данных, относящихся к определенному диапазону, достаточно лишь найти строку, в которой начинаются требуемые данные, после чего со всей вероятностью остальные искомые строки будут обнаружены на той же странице (иными словами, для достижения следующей строки почти не требуется усилий, поскольку все нужные строки хранятся вместе). С другой стороны, при использовании неупорядоченной таблице данные не являются связанными друг с другом каким-то иным способом, кроме как через индекс. Какая-либо физическая сортировка данных не осуществляется. Это означает, что для чтения данных из неупорядоченной таблицы системе может потребоваться обращаться для выборки строк к любым участкам файла с данными. В действительности вполне возможно (и даже не подлежит сомнению), что в процессе работы для получения данных придет-

ся неоднократно считывать в разное время одну и ту же страницу, поскольку в СУБД SQL Server нельзя предугадать, что снова придется обращаться к какому-то участку файла, так как между данными отсутствует связь. С другой стороны, если применяется кластеризованный индекс, то система может определить, что данные хранятся в отсортированном порядке и поэтому получить сразу все необходимые данные в результате чтения одной страницы.

Проводя сравнение некластеризованных индексов, заданных на неупорядоченной таблице, с кластеризованными индексами, ради справедливости необходимо отметить следующее: весьма велики шансы того, что многие считанные страницы будут оставаться в памяти (в кэше) и поэтому выборка других строк, относящихся к этим страницам, будет осуществляться исключительно быстро. Тем не менее и при таких обстоятельствах для выборки данных приходится выполнять несколько дополнительных логических операций.

На рис. 9.6 показана такая же процедура поиска, которая выполнялась с помощью кластеризованного индекса (см. рис. 9.5), но на этот раз с использованием некластеризованного индекса, заданного на неупорядоченной таблице.

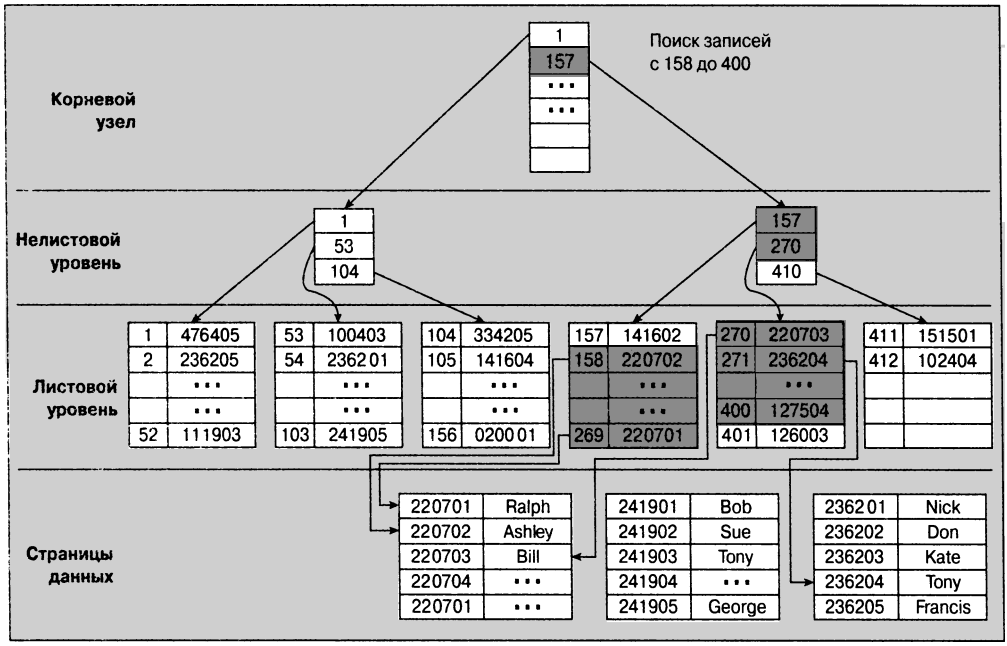


Рис. 9.6. Процедура поиска данных с помощью некластеризованного индекса, заданного на неупорядоченной таблице

При этом основная часть операций перемещения по индексу осуществляется точно так же, как и в предыдущем примере. Поиск начинается с того же корневого узла, после чего происходит переход вниз по уровням дерева, и в процессе этого перехода каждый раз обнаруживаются страницы, охватывающие все меньшую область представления данных, до тех пор, пока не достигается листовой уровень индекса. Различия обнаруживаются только начиная с этого момента. Если используется кластеризованный индекс, то поиск может быть остановлен непосредственно в этом

месте, а при использовании некластеризованного индекса приходится выполнять дополнительную работу. Если некластеризованный индекс задан на неупорядоченной таблице, то достаточно просто перейти на один уровень вниз. Для этого применяется идентификатор строки, полученный на странице индекса листового уровня, затем осуществляется переход к указанной строке, и только после этого достигается возможность обратиться к реальным данным.

Некластеризованные индексы, заданные на кластеризованной таблице

Если применяются **некластеризованные индексы, заданные на кластеризованной таблице**, то аналогии с кластеризованными индексами продолжаются, но обнаруживаются и свои различия. В некластеризованных индексах, заданных на кластеризованной таблице, так же, как и в некластеризованных индексах, заданных на неупорядоченной таблице, уровни индекса, отличные от листовых, во многом напоминают соответствующие уровни кластеризованного индекса. Различия обнаруживаются только после перехода к листовому уровню.

На листовом уровне обнаруживаются еще более резкие различия по сравнению с теми, которые обнаруживались между двумя другими индексными структурами, поскольку в индексе рассматриваемого типа для поиска данных применяется еще один индекс. При использовании кластеризованных индексов после достижения листового уровня обнаруживаются действительные данные. Если применяется некластеризованный индекс, заданный на неупорядоченной таблице, на листовом уровне обнаруживаются не искомые данные, а идентификатор, позволяющий перейти непосредственно к данным (т.е. для достижения данных нужно сделать всего лишь еще один шаг). А что касается некластеризованного индекса, заданного на кластеризованной таблице, то при его использовании на листовом уровне обнаруживается так называемый ***кластеризованный ключ****. Это позволяет получить достаточно информации для того, чтобы продолжить поиск, воспользовавшись кластеризованным индексом.*

Действия, осуществляемые при использовании некластеризованных индексов, заданных на кластеризованной таблице, показаны на рис. 9.7.

Таким образом, при использовании индекса рассматриваемого типа осуществляются две полностью разные процедуры поиска.

В примере, приведенном на рис. 9.7, вначале осуществляется поиск в диапазоне. Для этого выполняется один просмотр индекса, что позволяет выполнить выборку данных с помощью некластеризованного индекса для обнаружения непрерывного участка данных, соответствующих применяемому критерию (`LIKE 'T%'`). Просмотр такого рода, который позволяет перейти непосредственно к конкретному участку индекса, называется **позиционированием** (seek).

После этого начинается операция поиска второго типа — поиск с использованием кластеризованного индекса. Такая вторая операция поиска осуществляется с очень высоким быстродействием; единственная проблема заключается в том, что эта операция должна выполняться многократно. Дело в том, что в СУБД SQL Server после осуществления первой операции поиска с помощью индекса формируется список (в котором находятся все имена, начинающиеся с буквы "T"), но этот список логически не согласуется с кластеризованным ключом в виде каких-либо непрерывных участков, поэтому поиск каждой строки с данными должен выполняться отдельно, как показано на рис. 9.8.

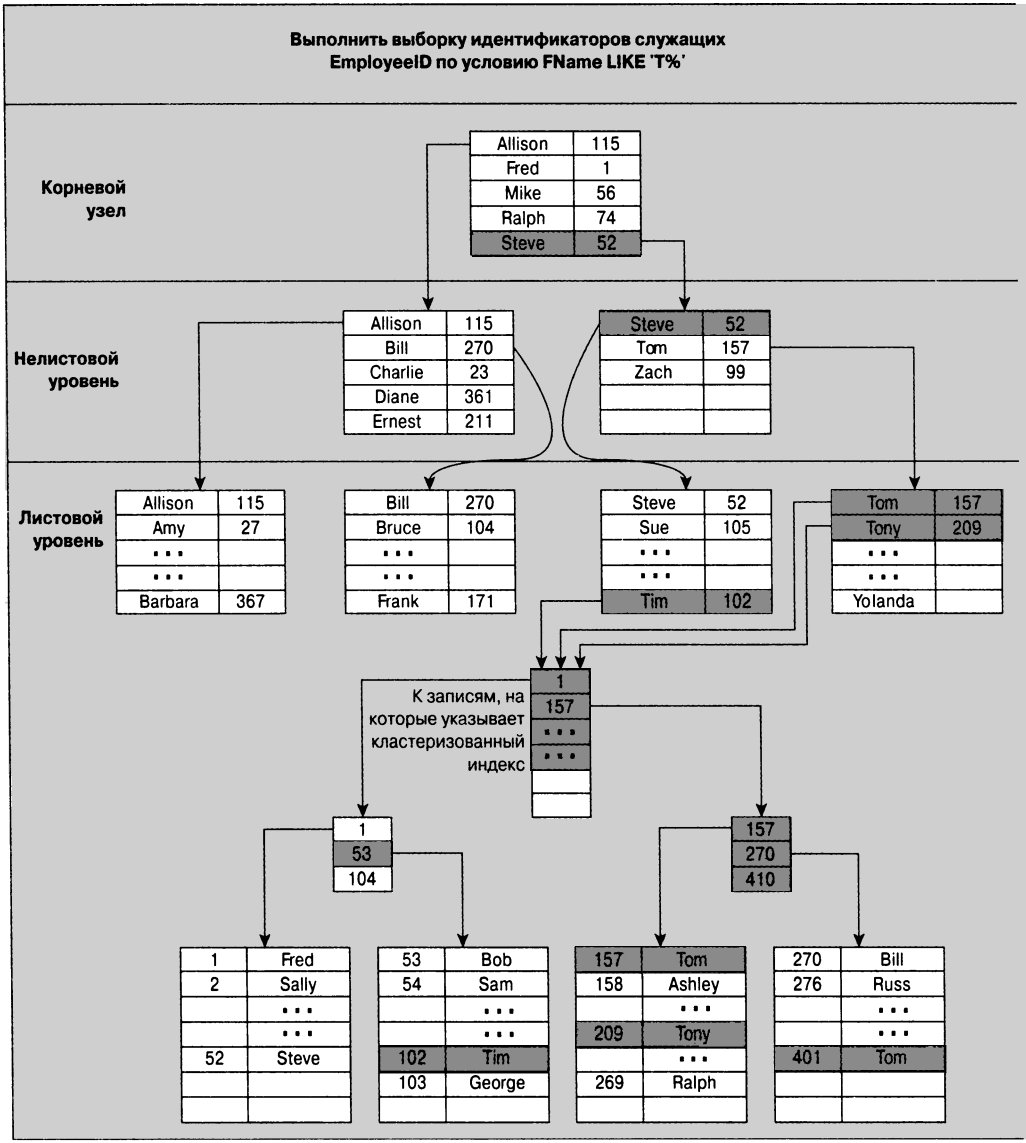


Рис. 9.7. Поиск данных с помощью некластеризованного индекса, заданного на кластеризованной таблице

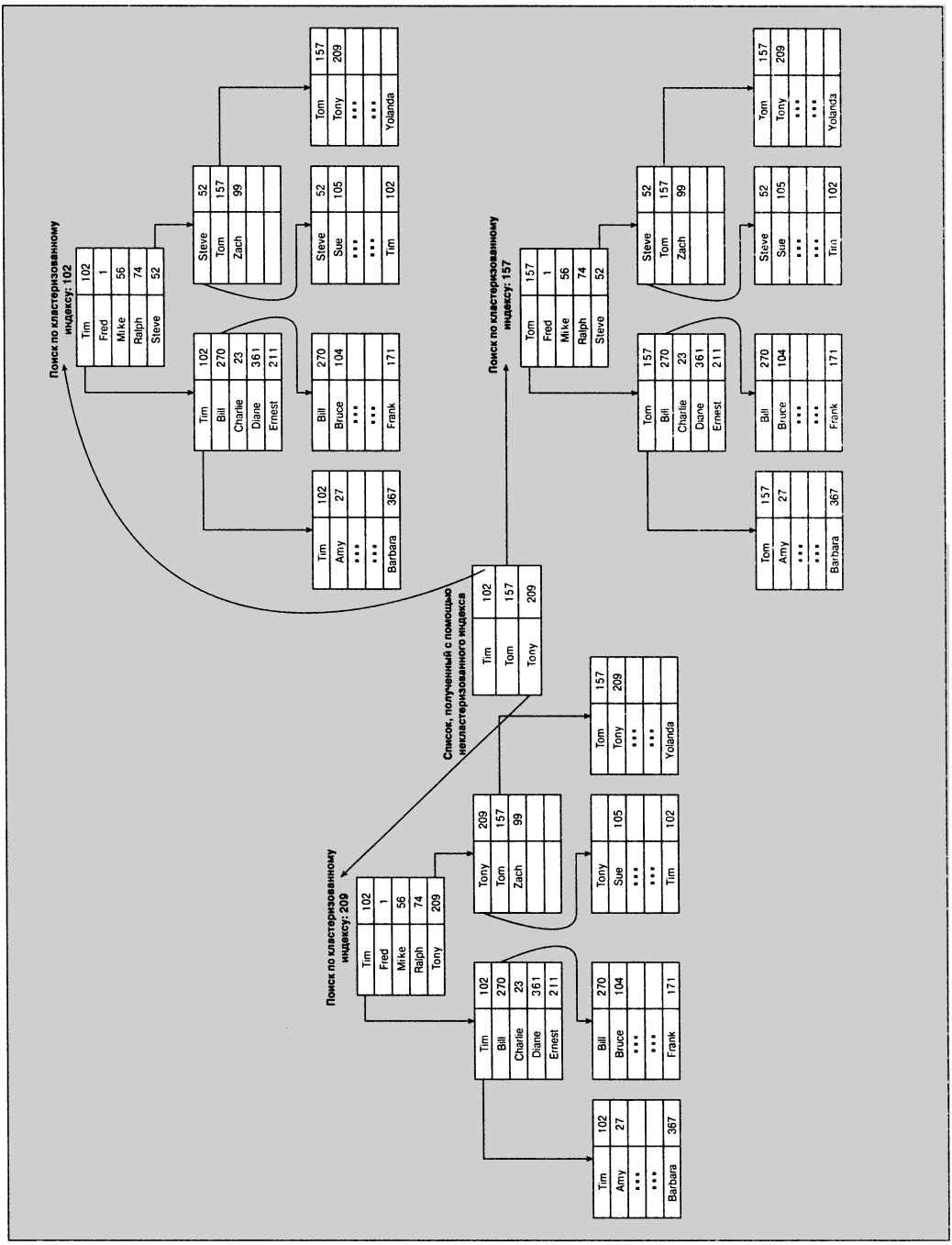


Рис. 9.8. Поиск отдельных строк с данными

Вполне очевидно, что в связи с необходимостью дважды выполнять поиск издержки становятся больше, чем в той ситуации, когда есть возможность использовать кластеризованный индекс с самого начала. А для первой операции поиска с помощью индекса (в которой применяется некластеризованный индекс) обычно требуется осуществить лишь немного логических операций чтения.

Например, если данные хранятся в таблице с длиной строки, равной 1000 байтов, и выполняется операция поиска, аналогичная показанной на рис. 9.8 (скажем, такая операция, которая должна возвратить 5 или 6 строк), то должно потребоваться примерно 8–10 логических операций чтения для получения информации из некластеризованного индекса. Но выполнение этих операций чтения позволяет лишь подготовиться к чтению строк с помощью кластеризованного индекса. На выполнение каждой такой операции поиска потребуется приблизительно 3–4 логических операций чтения, или 15–24 дополнительных операций чтения. На первый взгляд кажется, что такое увеличение количества операций не имеет особого значения, поэтому рассмотрим эту ситуацию под другим углом зрения.

Количество логических операций чтения увеличивается от минимального значения 3 до максимального значения 24, что соответствует увеличению объема работы, подлежащей выполнению, на 800%.

А теперь предположим, что масштабы обработки данных выросли и диапазон значений, подлежащих выборке с помощью некластеризованного индекса, составляет не такое небольшое значение, как пять или шесть строк, а пять или шесть тысяч строк или пять или шесть сотен тысяч строк; в таком случае влияние указанных факторов на производительность становится просто колоссальным.

Однако не следует придавать слишком большое значение тому, что применение некластеризованного индекса влечет за собой появление дополнительных издержек по сравнению с кластеризованным индексом. Соображения, изложенные в настоящем разделе, не направлены на то, чтобы убедить читателя не использовать некластеризованные индексы. Тем не менее следует учитывать то, что некластеризованные индексы при осуществлении операций чтения характеризуются более низкой эффективностью по сравнению с кластеризованными индексами (правда, в некоторых случаях некластеризованные индексы могут обеспечивать более высокую производительность при выполнении операций вставки). Кроме того, каким бы не был индекс, обычно он является основой наиболее быстрых способов поиска (хотя встречаются и исключения). Информация о том, какие индексы следует использовать в разных обстоятельствах и по каким причинам, приведена ниже.

Создание, модификация и удаление индексов

Операции создания, модификации и удаления индексов осуществляются во многом так же, как и применительно к другим объектам, таким как таблицы. В данном разделе рассматривается каждая из этих операций, начиная с операции создания.

Для создания индексов применяются два указанных ниже способа.

- Создание индекса с помощью явно заданной команды `CREATE INDEX`.
- Неявное создание индекса как обязательного объекта в результате ввода в действие некоторого ограничения.

Каждый из указанных способов имеет свои особенности, касающиеся того, что может или не может быть осуществлено с его помощью, поэтому рассмотрим каждый из них отдельно.

Оператор CREATE INDEX

Оператор CREATE INDEX осуществляет действие, полностью соответствующее его смыслу, — создает на указанной таблице или представлении индекс, основанный на заданных столбцах.

Синтаксическая структура оператора создания индекса является довольно разнообразной, и в ней используются некоторые элементы, которые фактически до сих пор не рассматривались в данной книге:

```
CREATE [UNIQUE] [CLUSTERED|NONCLUSTERED]
INDEX <index name> ON <table or view name>(<column name> [ASC|DESC] [,...n])
INCLUDE (<column name> [, ...n])
[WITH
[PAD_INDEX = { ON | OFF }]
[[,] FILLFACTOR = <fillfactor>]
[[,] IGNORE_DUP_KEY  = { ON | OFF }]
[[,] DROP_EXISTING = { ON | OFF }]
[[,] STATISTICS_NORECOMPUTE = { ON | OFF }]
[[,] SORT_IN_TEMPDB = { ON | OFF }]
[[,] ONLINE = { ON | OFF }
[[,] ALLOW_ROW_LOCKS = { ON | OFF }
[[,] ALLOW_PAGE_LOCKS = { ON | OFF }
[[,] MAXDOP = <maximum degree of parallelism>
]
[ON {<filegroup> | <partition scheme name> | DEFAULT }]
```

В последней версии SQL Server многие опции синтаксической конструкции оператора создания индекса изменились, поэтому варианты, применявшиеся в предыдущих версиях, теперь рассматриваются как устаревшие. Тем не менее во многих разработках встречаются случаи применения устаревшего синтаксиса в целях обеспечения совместимости с предыдущими версиями SQL Server. Однако устаревшие варианты синтаксических конструкций больше не поддерживаются и со временем их применение будет запрещено, поэтому автор настоятельно рекомендует при любой возможности стремиться использовать более новый синтаксис.

> Для создания индексов XML применяется аналогичный синтаксис, который, тем не менее, имеет существенные отличия. Информация по этой теме приведена в конце данного раздела.

Вообще говоря, синтаксическая структура оператора создания индекса соответствует такому же общему образцу, CREATE <object type> <object name>, как и многие другие операторы создания объектов, рассматривавшиеся до сих пор (а также напоминает те операторы создания объектов, о которых еще будет идти речь в настоящей книге). Специфика оператора создания индекса состоит в том, что в нем используется несколько промежуточных параметров, которые не встречаются в других операторах.

Так же как и в операторах создания представлений, которые будут рассматриваться в следующей главе, в операторе создания индекса приходится вводить дополни-

тельную конструкцию с учетом того факта, что индекс в действительности не представляет собой автономный объект. Индекс относится к некоторой таблице или к некоторому представлению, поэтому должно быть указано, к какому объекту относится столбец (столбцы), на котором задан индекс, с помощью ключевого слова "ON".

Обязательной является только конструкция ON <table or view name>(<column name>), а все остальные конструкции, которые следуют за ней, являются необязательными. Элементы синтаксического определения оператора создания индекса могут применяться в самых разных сочетаниях. Многие из этих элементов используются редко, а другие (такие как FILLFACTOR) могут оказывать существенное влияние на производительность системы и действия, осуществляемые в системе, поэтому ниже приведено последовательное описание различных опций.

Опция ASC/DESC

Ключевые слова ASC и DESC позволяют указывать, должен ли применяться в индексе порядок сортировки по возрастанию или по убыванию. По умолчанию применяется ключевое слово ASC (сокращение от ascending), которое, как и следовало ожидать, означает порядок сортировки по возрастанию.

На первый взгляд может показаться, что от выбора порядка сортировки по возрастанию или по убыванию мало что зависит, поскольку СУБД SQL Server позволяет просто выполнять просмотр в индексе в обратном порядке, если для поиска данных потребуется применение противоположного порядка сортировки. Тем не менее на практике выбор того или иного порядка сортировки влечет за собой более важные последствия. Просмотр индекса в обратном порядке осуществляется вполне успешно, если при этом приходится рассматривать только один столбец или если во всех используемых столбцах порядок сортировки остается всегда одинаковым, а если применяется индекс, в котором разные столбцы отсортированы по-разному, то ситуация изменяется. Иными словами, дела обстоят совсем иначе, если для поиска данных требуется, чтобы один столбец был отсортирован по возрастанию, а другой — по убыванию. Данные столбцов, включенных в состав индекса, хранятся вместе, поэтому переход к способу просмотра индекса для одного столбца в обратном порядке приводит также к изменению порядка следования данных в дополнительных столбцах на противоположный. Если же будет явно указано, что один столбец отсортирован по возрастанию, а другой — по убыванию, после чего произойдет изменение порядка следования данных во втором столбце на противоположный непосредственно в физически хранимых данных, то внезапно исчезает сама причина, по которой следовало бы перейти к использованию другого способа доступа к данным.

В качестве краткого примера рассмотрим такое задание по составлению отчета, по условиям которого необходимо упорядочить список служащих по дате найма на работу, начиная с принятого на работу в самую последнюю очередь (в порядке по убыванию), но требуется упорядочить этот список по фамилиям (в порядке по возрастанию). При использовании предыдущих версий SQL Server потребовалось бы выполнить две операции, относящиеся к первому и ко второму столбцам. А поскольку современная версия позволяет управлять порядком сортировки физически хранимых данных, обеспечивается свобода выбора способа комбинирования столбцов, применяемых в процессе обработки.

Вообще говоря, обычно не рекомендуется использовать опцию ASC/DESC (поскольку и в этом случае приходится учитывать проблемы обратной совместимости).

Тем не менее ниже указаны ситуации, в которых обычно приходится отступать от этой рекомендации.

- Существует необходимость в использовании разных вариантов порядка сортировки по возрастанию и по убыванию в различных столбцах.
- Проблема обеспечения обратной совместимости не является актуальной.

Опция INCLUDE

Опция INCLUDE представляет собой одно из самых удобных нововведений, впервые предложенных в версии SQL Server 2005. Назначение этой опции состоит в обеспечении лучшей поддержки так называемых охваченных запросов.

Опция INCLUDE позволяет указывать определенные столбцы в конструкции INCLUDE, а не указывать их только в конструкции ON. В таком случае СУБД SQL Server переносит содержимое указанных столбцов на листовой уровень индекса. Как уже было сказано, каждая строка на листовом уровне индекса соответствует строке данных, поэтому применение опции INCLUDE по существу равнозначно перемещению части исходных данных на листовой уровень индекса. Безусловно, изучение возможной области применения опции INCLUDE позволяет сделать вывод, что в действительности эту опцию имеет смысл применять только при создании некластеризованных индексов (кластеризованные индексы уже содержат данные на листовом уровне, поэтому для них нет смысла использовать рассматриваемую опцию).

Преимущества, которые могут быть получены при использовании опции INCLUDE, обусловлены тем, что в СУБД SQL Server применяется принцип, в соответствии с которым выполнение любой текущей операции прекращается сразу после того, как фактически достигается цель этой операции (о чем еще не раз будет идти речь в настоящей книге). Поэтому, если в СУБД SQL Server в процессе обработки индекса обнаруживается, что все необходимые данные могут быть получены без перехода к строке с действительными данными, то переход к выполнению операции чтения строки данных не выполняется (так как это бессмысленно). Включая в индекс конкретный столбец с помощью опции INCLUDE, можно "охватить" тот запрос, в котором используется данный конкретный индекс на листовом уровне, и обойтись без применения для перехода к странице с данными операций ввода-вывода, связанных с использованием указателя индекса.

Следует учитывать, что применение опции INCLUDE приводит и к отрицательным последствиям! Включение данных из отдельных столбцов в индекс с помощью опции INCLUDE влечет за собой увеличение размеров строк индекса листового уровня. Это вызывает уменьшение количества строк индекса, которые могут быть размещены на одной странице, поэтому для выборки одного и того же количества строк может потребоваться выполнение большего количества операций ввода-вывода. В результате может оказаться, что попытка ускорить выполнение одного запроса приведет к снижению быстродействия при выполнении других запросов. Поэтому необходимо выдерживать равновесие. Подумайте о том, какое влияние окажет применение опции INCLUDE на все компоненты вашей системы, а не только на тот конкретный запрос, который вы стремитесь реализовать в текущий момент.

Опция WITH

Опция WITH не требует особых пояснений, поскольку она применяется лишь для передачи СУБД SQL Server указания на то, что за ней последует одна или несколько дополнительных опций.

Опция PAD_INDEX

Среди опций, определяемых в списке WITH, опция PAD_INDEX находится на первом месте, но после ознакомления с тем, для чего предназначена опция PAD_INDEX, такой выбор места для ее определения кажется странным. Коротко можно отметить, что опция PAD_INDEX определяет лишь то, насколько полным должно быть заполнение страниц индекса нелистовых уровней (в процентах) при первоначальном создании индекса. Однако не следует задавать процентное значение опции PAD_INDEX, поскольку в качестве относительной величины заполнения используется процентное значение, заданное в опции FILLFACTOR, которая следует за ней. Применение значения параметра PAD_INDEX = ON не имеет смысла, если не задана опция FILLFACTOR (именно поэтому кажется неоправданным решение поместить опцию PAD_INDEX в начало списка опций).

Опция FILLFACTOR

При первоначальном создании индекса в СУБД SQL Server страницы по умолчанию заполняются настолько, насколько это возможно, за вычетом двух строк. В качестве значения опции FILLFACTOR можно задать любое число от 1 до 100. Это число показывает, насколько полным должно стать заполнение страниц в процентах после завершения процесса создания индекса. Но следует учитывать, что при разбиении страниц данные все равно перераспределяются между двумя страницами в равных долях. Это означает, что в процессе эксплуатации базы данных невозможно постоянно сохранять контроль над тем, в каком процентном отношении заполняются страницы индекса, иначе как путем регулярной перестройки индекса (а эту операцию действительно следует выполнять; информация о том. как подготовить график технического сопровождения для этой операции, приведена в главе 20).

Опция FILLFACTOR применяется в тех случаях, когда требуется откорректировать используемое значение плотности заполнения страниц. Рекомендации по выбору значения этой опции приведены ниже.

- В системе OLTP рекомендуется применять низкие значения FILLFACTOR.
- В системе OLAP или в другой очень статичной системе (под этим подразумевается неподверженность системы изменениям, т.е. применение очень малого количества операций вставки и удаления) значение FILLFACTOR должно быть максимально возможным.
- В системе с небольшим относительным количеством транзакций, но более значительным количеством запросов, предназначенных для получения отчетов, по-видимому, следует выбирать какое-то промежуточное значение (не слишком низкое и не слишком высокое).

Если значение опции FILLFACTOR не задано, то СУБД SQL Server полностью заполняет страницы, за вычетом двух строк; при этом минимальное количество строк составляет одну строку в расчете на одну страницу (например, если строка имеет длину 8000 символов, то на странице может поместиться только одна строка, поэтому условие, согласно которому должно оставаться свободное место для размещения двух строк, не применяется).

Опция IGNORE_DUP_KEY

Опция IGNORE_DUP_KEY позволяет добиться чуть большего, чем просто обойти ограничения системы. Кратко можно отметить, что при использовании этой опции ограничение UNIQUE оказывает немного другое воздействие, чем в противном случае.

При обычных обстоятельствах применение ограничения уникальности или уникального индекса приводит к тому, что не допускается вставка каких-либо дубликатов любого рода. В частности, если в транзакции предпринимается попытка создать дубликат на основе значения столбца, который определен как уникальный, то подобная попытка отвергается и производится откат транзакции. Но если задана опция IGNORE_DUP_KEY, то система действует немного иначе. Сообщение об ошибке все еще формируется, но возникающая при этом ошибка относится только к уровню предупреждения; тем не менее вставка строки все равно не выполняется.

С точки зрения применения опции IGNORE_DUP_KEY последняя особенность, согласно которой вставка строки все равно не выполняется, имеет принципиальную важность. Дело в том, что дублирующаяся строка все равно отвергается, но для транзакции не формируется команда на выполнение отката (поскольку ошибка формируется как вызывающая предупреждающее сообщение, а не сообщение о критической ошибке).

Очевидно, что опция IGNORE_DUP_KEY может применяться как способ обеспечения вставки уникальных значений, применение которого, тем не менее, не приводит к нарушению работы транзакции, в которой предпринимается попытка вставить дубликат. Каковым бы ни был процесс, в котором предпринимаются попытки вставить строку, рассматриваемую как дубликат, вполне возможно, с точки зрения организации этого процесса фактически не имеет значения то, что эта строка — дублирующаяся (в связи с этим не возникают какие-либо логические ошибки). Это означает, что функционирование процесса может быть организовано таким образом, что для него может иметь значение лишь наличие строки с определенным содержимым, независимо от того, была ли выполнена вставка искомой строки в этом или в другом процессе.

Опция DROP_EXISTING

Если задана опция DROP_EXISTING, то перед созданием нового индекса удаляется существующий индекс с именем, которое совпадает с именем создаваемого индекса. Эта опция обеспечивает гораздо более эффективное формирование индекса по сравнению с тем, когда происходит просто удаление и повторное создание существующего индекса, если он используется в сочетании с кластеризованным индексом. Если в результате перестройки должен быть получен индекс, полностью совпадающий с существующим индексом, то СУБД SQL Server определяет, что такой некластеризованный индекс не следует подвергать операции удаления и повторного создания, а при выполнении явно заданных операций удаления и создания поневоле приходится дважды выполнять перестройку всех некластеризованных индексов с учетом изменения местонахождения строк. Если же структура индекса модифицируется с применением опции DROP_EXISTING, то перестройка некластеризованных индексов осуществляется не два раза, а только один раз. Более того, невозможно просто удалить и повторно создать индекс, сформированный с помощью ограничения, например, чтобы ввести в действие определенное значение коэффициента заполнения. А опция DROP_EXISTING позволяет обойти это ограничение.

Опция STATISTICS_NORECOMPUTE

По умолчанию в СУБД SQL Server предпринимаются попытки автоматизировать процесс обновления статистических данных, относящихся к используемым таблицам и индексам. Выбирая опцию STATISTICS_NORECOMPUTE, вы утверждаете, что берете ответственность за обновление статистических данных на себя. А для того чтобы отменить эту опцию, необходимо вызвать на выполнение команду UPDATE STATISTICS, но не использовать повторно опцию STATISTICS_NORECOMPUTE.

Автор настоятельно рекомендует не использовать эту опцию. Дело в том, что статистические данные, относящиеся к индексу, используются оптимизатором запросов для определения того, насколько данный индекс способствует повышению производительности выполнения данного конкретного запроса. Статистические данные, касающиеся индекса, постоянно изменяются по мере увеличения и уменьшения объема данных, хранящихся в таблице, а также изменения конкретных значений в столбцах. Сопоставив эти два факта, можно легко понять, что отказ от автоматического обновления статистических данных приведет к тому, что оптимизатор запросов будет выбирать способы выполнения запросов из устаревшей информации. А если средство автоматического обновления статистических данных останется действующим, это приведет к регулярному обновлению статистических данных (точные сведения о том, как часто будет происходить такое обновление, зависят от характера и частоты обновления таблицы). С другой стороны, отмена опции автоматического обновления статистических данных приводит к тому, что данные, применяемые оптимизатором запросов, становятся устаревшими или возникает необходимость ввести в действие график вызова на выполнение команды UPDATE STATISTICS вручную.

Опция SORT_IN_TEMPDB

Применение опции SORT_IN_TEMPDB имеет смысл, только если база данных tempdb хранится на физическом жестком диске, отличном от того, где хранится база данных, в состав которой должен войти новый индекс. Функция распределения файлов по жестким дискам в основном относится к области деятельности администратора базы данных, поэтому в настоящем разделе будет дан лишь краткий обзор того, что означает эта опция и почему имеет смысл размещать базу данных tempdb на отдельном физическом устройстве.

При формировании индекса в СУБД SQL Server выполняется целый ряд операций чтения, описанных ниже, для осуществления различных этапов создания индекса.

1. Чтение всех данных таблицы и создание листовой строки, соответствующей каждой строке действительных данных. Информация листовой строки, так же как действительные данные и окончательно сформированный индекс, сохраняется на страницах, предназначенных для промежуточной обработки. Эти промежуточные страницы не представляют собой окончательно сформированные страницы индекса и служат скорее в качестве места для временного сохранения данных после каждого заполнения буферов сортировки.
2. Выполнение отдельного прогона по промежуточным страницам для их слияния и преобразования в окончательно сформированные страницы листового уровня индекса.
3. Создание все новых и новых нелистовых страниц по мере заполнения листовых страниц.

Если опция SORT_IN_TEMPDB не используется, то промежуточные страницы записываются на тот же жесткий диск, где находятся физические файлы, в которых хранится база данных. Это означает, что происходит конкуренция между операциями чтения действительных данных и операциями записи, осуществляемыми в процессе создания индекса. Кроме того, для выполнения этих двух типов операций головки жесткого диска должны перемещаться с одного места на другое (поскольку чтение осуществляется в одном файле, а запись — в другом). В результате этого продолжительность выполнения операций чтения и записи увеличивается.

Если, с другой стороны, используется опция SORT_IN_TEMPDB, то промежуточные страницы записываются в базу данных tempdb, а не в файл самой базы данных. Кроме того, если пользовательская база данных и база данных tempdb находятся на разных физических дисках, то, безусловно, конкуренция между операциями доступа к пользовательской базе данных и операциями формирования индекса полностью исключается. Однако следует учитывать, что указанное преимущество достигается, только если база данных tempdb находится на физическом диске, отличном от того, где находится пользовательская базы данных; в противном случае изменяется только имя файла, в котором записываются промежуточные данные, и конкуренция между операциями ввода-вывода продолжает оставаться важным отрицательным фактором.

Прежде чем приступить к использованию опции SORT_IN_TEMPDB, убедитесь в том, что в базе данных tempdb имеется достаточный объем пространства для формирования индексов на используемых таблицах.

Опция ONLINE

Если для опции ONLINE задано значение ON, то принудительно устанавливается такой режим доступа к индексируемой таблице, что эта таблица остается применимой для общего доступа и не создаются какие-либо блокировки, которые не позволяли бы пользователям обращаться к индексу и (или) таблице. По умолчанию при выполнении операций создания полного индекса происходит захват блокировок, необходимых для получения полного и эффективного доступа к таблице (что в конечном итоге приводит к блокировке всей таблицы). Но установка таких блокировок имеет побочный эффект, заключающийся в том, что пользователи теряют на время возможность работать с таблицей (в этом действительно заключается парадокс, поскольку индекс, скорее всего, создается для повышения удобства работы с базой данных, но на время осуществления операции создания индекса таблица становится неприменимой).

На первый взгляд кажется, что обеспечение с помощью опции ONLINE доступа к таблице, на которой создается индекс, — весьма неплохая идея, но фактически дело обстоит иначе. Следует учитывать, что любые операции создания индекса, в том числе выполняемые с опцией ONLINE, требуют выполнения очень большого объема операций ввода-вывода, поэтому производительность работы пользователей в этот период так или иначе будет снижаться. Кроме того, при использовании опции ONLINE возникает значительный объем дополнительных издержек, связанных с предотвращением таких ситуаций, в которых кто-либо из пользователей получил бы из базы данных неправильную информацию. Если СУБД SQL Server имеет возможность приобрести полную власть над таблицей на время создания индекса, то индекс формируется намного быстрее и сокращается общая продолжительность времени, в течение которого процесс формирования индекса отрицательно отражается на работе системы.

Операции формирования индекса с использованием опции ONLINE поддерживаются только в версии Enterprise Edition программного обеспечения SQL Server. Безусловно, операция создания индекса с опцией ONLINE может быть вызвана на выполнение и при использовании других версий, но требование по предоставлению пользователям доступа к таблице игнорируется, поэтому, если применяется менее мощная версия SQL Server, не следует удивляться, вызвав на выполнение операцию создания индекса с опцией ONLINE и обнаружив, что в связи с этим работа всех пользователей остановлена.

Опции ALLOW_ROW_LOCKS и ALLOW_PAGE_LOCKS

Опции ALLOW_ROW_LOCKS и ALLOW_PAGE_LOCKS, в отличие от опции ONLINE, были введены в состав оператора CREATE INDEX в более ранних версиях. Тема, касающаяся применения этих опций, является чрезвычайно сложной. Поскольку настоящая книга предназначена для начинающих, а приведенное в ней до сих пор описание блокировок таблиц является очень кратким, остановимся на довольно простом пояснении.

В данной книге уже неоднократно использовался термин "блокировка". Как было описано выше, блокировка – это своего рода способ резервирования ресурсов, позволяющий предотвратить конфликты между операциями доступа к данным, которые могли бы привести к нарушению целостности данных. Опции ALLOW_ROW_LOCKS и ALLOW_PAGE_LOCKS служат для указания на то, могут ли применяться для создаваемого индекса блокировки на уровне страницы или на уровне строки. Использование этих опций может оказывать весьма заметное влияние на производительность, поэтому к ним следует прибегать с исключительной осторожностью.

Опция MAXDOP

Опция MAXDOP позволяет переопределить значение параметра настройки конфигурации системы, определяющее максимальную степень распараллеливания (Degree Of Parallelism – DOP), которая применяется при формировании индекса. В настоящей книге еще не было ничего сказано о распараллеливании операций, поэтому в данном разделе будут приведены некоторые сведения по этой теме.

Коротко можно отметить, что степень распараллеливания определяет максимальное количество процессов, которые могут быть введены в действие в целях осуществления одной операции в базе данных (в рассматриваемом случае речь идет о формировании индекса). В системе предусмотрен параметр настройки конфигурации, определяющий максимальную степень распараллеливания, который позволяет установить максимальное количество процессов, используемых для выполнения одной операции. А опция MAXDOP, которая входит в состав опций создания индекса, позволяет задать более высокую или более низкую степень распараллеливания по сравнению со значением базового параметра системы в соответствии с конкретными потребностями.

Опция ON

В СУБД SQL Server предусмотрена возможность обеспечить хранение индексов отдельно от данных с помощью опции ON. Такая возможность является очень удобной по многим причинам, в том числе указанным ниже.

- Пространство, требуемое для хранения индексов, может быть распределено по другим жестким дискам.
- Операции ввода-вывода, связанные с обработкой индекса, не создают дополнительную нагрузку, которая препятствовала бы выполнению операций физической выборки данных.

Есть и другие важные соображения в пользу хранения индексов отдельно от данных, но они относятся к области чрезвычайно сложной тематики. Чтобы можно было содержательно раскрыть эту тему, необходимо рассмотреть большой объем сведений, касающихся способов хранения и применения данных, но, по мнению автора, изложение этих сведений выходит за рамки настоящей книги.

Создание индексов XML

Индексы XML были впервые введены в версии SQL Server 2005, и автор должен признать, что был приятно поражен тем, что корпорация Microsoft выполнила задачу по внедрению этих индексов. Я давно знаю многих представителей коллектива разработчиков Microsoft и испытываю к ним глубокое доверие, но задача индексации кода с такой сложной структурой, как код XML, всегда представляла собой проблему, которую многие пытались решить, но лишь немногие сумели добиться реального успеха. Хочу выразить глубокую благодарность коллективу разработчиков программного обеспечения SQL Server за то, что сумели справиться с этой задачей. Подробные сведения об индексах XML и их использовании приведены ниже.

При изложении этой темы автору снова приходится сталкиваться с тем, что сведения, необходимые для полного ее понимания, еще не приведены, поскольку в настоящей книге проблематика XML еще до сих пор фактически не рассматривалась. Однако я считаю, что сведения об индексах XML в большей степени относятся к проблематике индексов, а не к проблематике XML. И действительно, в синтаксическом определении оператора создания индекса XML поддерживаются те же опции, которые рассматривались в приведенном выше описании `CREATE INDEX`, за исключением опций `IGNORE_DUP_KEY` и `ONLINE`. Поэтому ниже приведено чрезвычайно краткое предварительное описание.

В отличие от реляционных данных, которые рассматривались до сих пор в настоящей книге, данные, представленные в коде XML, обычно имеют гораздо более сложную структуру. В коде XML для идентификации данных используются дескрипторы, а с самим документом XML может быть связана так называемая *схема*, позволяющая предоставить информацию о типе и структуре, которая может применяться для проверки допустимости данных, оформленных с помощью кода XML. А в связи с тем, что документы XML имеют сложную структуру, для поиска узлов с данными в документе XML должен быть предусмотрен способ перехода к нужному узлу документа, или способ предоставления информации о пути такого перехода. С другой стороны, индексы предназначены для жесткой регламентации и очень точного определения структуры и порядка расположения данных; в этом и состоит вся сложность решения задачи определения индексов для документа XML.

В версии SQL Server 2005 предусмотрена возможность создавать индексы на столбцах, предназначенных для хранения данных типа XML. При этом должны учитываться описанные ниже основные требования.

- На таблице, содержащей код XML, который подлежит индексации, должен быть задан кластеризованный индекс.
- Прежде чем появится возможность создавать вторичные индексы XML, на столбце с данными XML должен быть определен первичный индекс XML (дополнительная информация по этой теме приведена ниже).
- Индексы XML должны создаваться только на столбцах типа XML (причем индекс XML является единственным типом индекса, который может создаваться на столбцах, содержащих данные такого типа).
- Столбец с данными XML должен входить в состав базовой таблицы, поскольку индекс XML не может быть создан на представлении.

Первичный индекс XML

Первый индекс, создаваемый на столбце с данными XML, должен быть объявлен как первичный (primary). При создании первичного индекса в СУБД SQL Server формируется новый кластеризованный индекс, в котором объединяются кластеризованный индекс базовой таблицы и данные из указанного узла XML.

Вторичные индексы XML

Вторичные индексы XML не имеют каких-либо важных отличительных особенностей. Во многом аналогично тому, как некластеризованные индексы указывают на кластеризованный ключ кластеризованного индекса, вторичные индексы XML указывают на первичный индекс XML в основном по такому же принципу. После создания первичного индекса XML на том же столбце с данными XML может быть создано дополнительно до 248 вторичных индексов XML.

Подразумеваемые индексы, которые создаются после ввода в действие ограничений

Автор хотел бы внести предложение называть индексы, создаваемые в связи с вводом в действие ограничений, "индексами, создаваемыми по случаю". Этим я не хочу сказать, что такого индекса не должно быть; речь идет лишь о том, что причиной создания индекса становится ввод в действие ограничения. Кроме того, я применяю такую формулировку, поскольку в своей работе очень часто сталкивался с ситуациями, когда единственными индексами в системе были только созданные таким образом. Обычно при анализе подобных ситуаций вполне оправдывается предположение, что администраторы и (или) проектировщики системы фактически не учитывают, что применение индексов весьма способствует повышению производительности системы.

Но иногда в области применения индексов наблюдается другие, не менее странные отклонения. Таковыми являются ситуации, в которых администратор или проектировщик признает важность индексов, умеет их создавать, но фактически не задумывается над тем, сколько индексов уже определено в системе и для чего они предназначены. Ситуации подобного рода характеризуются также тем, что в них применяются индексы, дублирующие друг друга. При условии, что для индексов используются разные имена, СУБД SQL Server не препятствует созданию одинаковых индексов.

Подразумеваемые индексы создаются после определения для таблицы одного из двух перечисленных ниже ограничений.

- Ограничение PRIMARY KEY.
- Ограничение UNIQUE (называемое также определением **альтернативного ключа**).

Выше в данной главе синтаксическая структура оператора CREATE INDEX рассматривалась достаточно подробно, поэтому мы не будем возвращаться к этой теме. Но следует отметить, что при создании индекса, который рассматривается как подразумеваемый индекс, относящийся к ограничению, не допускается применение любых опций, кроме {CLUSTERED|NONCLUSTERED} и FILLFACTOR.

Обоснованное принятие решения о том, где и когда должны использоваться индексы

Приведенные выше сведения позволяют сделать вывод о том, что следует всегда создавать кластеризованные индексы. И действительно, можно привести целый ряд соображений в пользу такого мнения. Однако следует учитывать, что в определенных обстоятельствах подобное решение становится неоправданным.

Достаточно сложной является даже сама задача принятия решения о том, какие индексы должны быть созданы в базе данных. а в связи с необходимостью выбирать при этом также типы индексов, эта задача становится еще более затруднительной. Принятие решения о выборе типа индекса становится одновременно и проще, и сложнее в связи с тем, что на таблице может быть задан только один кластеризованный индекс. Прежде всего необходимо тщательно взвесить все факторы, свидетельствующие в пользу этого решения или позволяющие определить, что оно является неприемлемым.

Избирательность

Индексы, особенно некластеризованные, обеспечивают наиболее существенное повышение производительности в основном в таких ситуациях, когда с помощью индекса может быть достигнут достаточно высокий уровень **избирательности**. Избирательностью называется относительное количество уникальных значений в столбце. Чем выше процентная доля уникальных значений в столбце, тем выше избирательность и тем значительнее повышение производительности благодаря индексам.

Как уже было сказано в разделах с описанием некластеризованных индексов (особенно некластеризованных индексов, заданных на кластеризованном индексе), поиск в некластеризованном индексе фактически является первым этапом обеспечения доступа к данным. Для того чтобы найти требуемые данные, необходимо выполнить еще одну операцию поиска, но на этот раз с помощью кластеризованного индекса. Даже при использовании некластеризованного индекса, заданного на неупорядоченной таблице, все равно приходится в конечном итоге выполнять несколько отдельных физических операций чтения.

Если для получения доступа к данным вслед за выполнением одной операции поиска в некластеризованном индексе приходится выполнять еще несколько дополнительных операций поиска в кластеризованном индексе, то, по-видимому, лучше

прибегнуть к полному просмотру таблицы. В противном случае по мере уменьшения избирательности ключа происходит удивительно быстрый, почти экспоненциальный рост количества выполняемых операций. Есть основания полагать, что при наличии всего лишь 90–95% уникальных значений в индексированном столбце нет смысла использовать некластеризованный индекс, поскольку в связи с циклической организацией процесса доступа к данным создаются весьма значительные издержки.

Кластеризованные индексы в значительно меньшей степени подвержены отрицательному влиянию низкой избирательности, поскольку позволяют сразу же переходить на начало участка таблицы с требуемыми данными, независимо от того, является ли ключ уникальным или нет, а после этого поиск необходимой строки осуществляется очень просто. К тому же не требуется чтение дополнительных страниц индекса. Кроме того, весьма велика вероятность, что кластеризованный индекс поможет также упростить поиск за счет других предоставляемых им возможностей.

Безусловно, внешние ключи чаще всего используются на стороне "многие" в связи "один ко многим", поэтому, казалось бы, на столбцах внешних ключей также нет смысла создавать индексы, поскольку они по определению имеют низкую избирательность. Но дело обстоит иначе, поскольку внешние ключи часто применяются в операциях соединения с таблицей, на которую они ссылаются. А применение индексов, независимо от избирательности столбцов, на которых они заданы, весьма способствует повышению производительности соединения, в связи с этим с помощью индексов могут выполняться так называемые ***соединения слиянием****. При выполнении операции соединения слиянием происходит выборка строк каждой таблицы и их сравнение для определения их соответствия критериям соединения (условиям, лежащим в основе соединения). А поскольку индексы заданы на связанных столбцах в обеих таблицах, поиск строк происходит очень быстро.*

Из этого следует, что такой критерий, как избирательность, не является решающим, но остается все же довольно важным. Если рассматриваемый столбец не используется в качестве внешнего ключа, то для принятия обоснованного решения о том, следует ли задать на нем индекс, необходимо в первую очередь определить, как часто он будет использоваться, а после этого оценить его избирательность.

Учет затрат на сопровождение индексов

Следует помнить, что индексы, повышая производительность операций чтения данных, вместе с тем требуют очень больших издержек во время модификации данных. Выполнение операций обновления, удаления и вставки данных влечет за собой необходимость сопровождения индексов. После каждого внесения изменений в данные необходимо также обновлять все индексы, относящиеся к этим данным.

После вставки новой строки в таблицу необходимо также ввести по одной новой строке в каждый индекс, заданный на этой таблице. Следует также помнить, что операция обновления строки осуществляется с помощью двух операций, удаления и вставки, поэтому также приходится обновлять индексы. Но этого еще не достаточно! После удаления строк не только изменяется состав хранимых в таблице данных; приходится обновлять все индексы. Таким образом, создание каждого нового индекса приводит к увеличению потенциального количества строк, которые приходится обновлять.

В предыдущем абзаце не случайно речь зашла о строках, а не об одной строке. Напомним, что B-дерево состоит из нескольких уровней. После каждого внесения изменений на листовом уровне возникает вероятность того, что произойдет разбиение

страницы или потребуется внести изменения в одну или несколько страниц нелистовых уровней в целях корректировки ссылок, для того чтобы они правильно указывали на страницу листового уровня.

Иногда (а фактически очень часто) наилучшим решением становится отказ от создания еще одно индекса. А в некоторых обстоятельствах лучше всего ограничиться созданием индексов с учетом требований поддержки транзакций, наиболее важных для прикладной системы, в которых используется рассматриваемая таблица. При этом необходимо учитывать, имеется ли в коде транзакции конструкция `WHERE`, какой столбец (столбцы) в ней используется, и требуется ли сортировка.

Определение условий применения кластеризованного индекса

Следует учитывать, что на таблице может быть задан только один кластеризованный индекс, поэтому выбирать его нужно очень тщательно.

По умолчанию во время создания кластеризованного индекса создается также первичный ключ. Чаще всего такое сочетание определений кластеризованного индекса и первичного ключа является наиболее приемлемым, но так бывает не всегда (а в некоторых случаях реализация такого решения может привести к заметному снижению производительности). Если в подобных ситуациях не будут проведены определенные действия, то окажется, что кластеризованный индекс больше нельзя применить для каких-либо других целей. В этом случае лучше всего отказаться от использования решения, предусмотренного по умолчанию. Иными словами, следует исходить из того, какой первичный ключ является наиболее приемлемым, и оценить, действительно ли он должен быть создан на основе кластеризованного индекса.

Если будет обнаружено, что должен быть действительно принят другой подход, иначе говоря, если будет определено, что первичный ключ не должен быть задан на кластеризованной таблице, достаточно ввести ключевое слово `NONCLUSTERED` при создании таблицы, например, следующим образом:

```
CREATE TABLE MyTableKeyExample
(
  Column1   intIDENTITY
    PRIMARY KEY NONCLUSTERED,
  Column2   int
)
```

После создания индекса единственный способ внести в него изменения состоит в том, чтобы удалить его и снова сформировать, поэтому необходимо стремиться к тому, чтобы можно было с самого начала правильно задать используемый индекс.

Следует учитывать, что после внесения изменений в столбец (столбцы), на котором определен кластеризованный индекс, в СУБД SQL Server может потребоваться выполнение полной пересортировки всей таблицы (напомним, что при использовании кластеризованного индекса порядок сортировки строк таблицы и расположения строк кластеризованного индекса являются одинаковыми). Итак, если предположить, что кластеризованный индекс задан на таблице, которая состоит из строк длиной 5000 символов и включает в себя миллион строк, то становится очевидно, что иногда объем данных, требующих переупорядочения, становится колоссальным. В связи с этим необходимо найти ответ на несколько приведенных ниже вопросов.

- Много ли времени потребуется для переупорядочения данных в таблице? Для выполнения этой операции может потребоваться много времени, но удобный способ оценки затрат времени на выполнение указанной операции фактически отсутствует.
- Имеется ли достаточный объем свободного пространства? Следует учитывать, что для переупорядочения таблицы с кластеризованным индексом в среднем требуется дополнительный объем, превышающий в 1,2 раза объем пространства, которое в настоящее время занимает таблица (при этом учитывается необходимость распределения рабочего пространства и нового индекса). Поэтому, если обрабатывается очень большая таблица, то может потребоваться весьма существенный объем дополнительного пространства, поэтому нужно будет подготовить место для его распределения. Между прочим, все названные операции осуществляются в самой базе данных, поэтому возможность переупорядочения любой кластеризованной таблицы определяется тем, какие значения опций максимального размера и роста заданы для базы данных.
- Следует ли использовать опцию SORT_IN_TEMPDB? Если предусмотрено размещение базы данных tempdb на физическом жестком диске, отличном от того, где находится основная база данных, и на этом жестком диске имеется достаточный объем пространства, то ответ на этот вопрос, по-видимому, должен быть положительным.

Преимущества кластеризованных индексов

Кластеризованные индексы целесообразно задавать на таких таблицах, в которых рассматриваемый столбец (столбцы) часто применяется в таких запросах, которые охватывают целый ряд строк. Отличительной особенностью подобных запросов является то, что в них используются операции BETWEEN или операции сравнения < и >. Характерными примерами запросов, в которых осуществляется доступ к целому ряду строк, поэтому для их выполнения хорошо подходят кластеризованные индексы, являются такие запросы, в которых применяются конструкции GROUP BY и агрегирующие функции MAX, MIN и COUNT. Кластеризация способствует успешному выполнению этих запросов, поскольку обеспечивается возможность начать поиск с определенной строки в составе хранимых данных, затем продолжать чтение до тех пор, пока остаются в силе условия, по которым должна быть выполнена выборка данных, а после этого прекратить обработку данных. Такая организация доступа к данным является чрезвычайно эффективной.

Кроме того, кластеризованные индексы обеспечивают исключительно высокую производительность, если данные должны быть получены в отсортированном виде с помощью конструкции ORDER BY, в которой используется кластеризованный ключ.

Недостатки кластеризованных индексов

Создание кластеризованного индекса на каком-то определенном столбце (столбцах) может стать нецелесообразным по двум описанным ниже причинам. Основная причина вполне очевидна — наличие лучшего места для его использования. Необходимо еще раз отметить, что не следует задавать кластеризованный индекс на том или ином столбце лишь потому, что такое решение на первый взгляд кажется целесообразным (практика показывает, что чаще всего без раздумий принимают решение о

создании кластеризованного индекса на столбце первичного ключа). Следует вначале убедиться в том, что действительно выбран наиболее подходящий столбец.

Но, по-видимому, гораздо более серьезной причиной отказа от использования кластеризованных индексов является перспектива того, что в таблице будет применяться большое количество таких операций вставки, в которых строки не заданы последовательно. Напомним, что при этом повышается вероятность возникновения разбиений страниц, а выполнение таких операций связано со значительными затратами времени.

Рассмотрим описанную ниже ситуацию. Предположим, что ведется разработка системы бухгалтерского учета и принято решение использовать номер транзакции (термин *транзакция* в данном контексте обозначает ряд взаимосвязанных бухгалтерских операций) в качестве первичного ключа в таблице учета транзакций. Тем не менее в процессе разработки системы обнаруживается, что желательно было бы предусмотреть включение в номера транзакций каких-либо обозначений, позволяющих узнать, к какой категории относится транзакция (к тому же применение таких обозначений может существенно помочь бухгалтерам, эксплуатирующим систему, при выявлении ошибок). Поэтому разработчик вносит предложение — вводить во все номера транзакций префикс, указывающий, к какой подсистеме относится данная транзакция. В конечном итоге номера транзакций принимают примерно такой вид, где XXXXXX представляет собой последовательное числовое значение:

```
ARXXXXXX          Accounts Receivable Transactions
GLXXXXXX          General Ledger Transactions
APXXXXXX          Accounts Payable Transactions
```

Разработчик находит эту идею великолепной и приступает к ее реализации, не пересматривая применяемую по умолчанию опцию, согласно которой кластеризованный индекс задается на первичном ключе.

На первый взгляд решение о применении ключа с указанной структурой кажется вполне приемлемым. Все строки в таблице будут обозначаться уникальными ключами, а бухгалтеры смогут с успехом использовать возможность извлекать дополнительную информацию из номера транзакции. Кроме того, кажется приемлемым кластеризованный индекс, поскольку он позволяет успешно выполнять часто применяемые запросы, в которых предусмотрена обработка ряда последовательных значений номеров транзакций.

В действительности применение указанной структуры номеров транзакций приводит отнюдь не к таким благоприятным последствиям. Достаточно лишь рассмотреть, как будут выполняться операции вставки. Кластеризованный индекс с самого начала предназначается для использования в качестве превосходного механизма, позволяющего избежать возникновения основной части издержек, связанных с разбиением страниц. Ведь если вставка новой строки должна осуществляться так, что ее место расположения будет находиться вслед за последней строкой в таблице, то даже в случае разбиения страницы на новую страницу должна переходить единственная, вновь вставляемая строка, поэтому в СУБД SQL Server не приходится осуществлять какие-либо попытки перемещения существующих данных. Но при использовании указанной организации данных возникает иная картина.

Новые строки, относящиеся к главной книге (General Ledger — GL), действительно будут записываться вслед за последней строкой таблицы (поскольку префикс GL занимает после сортировки по алфавиту последнее место в списке префиксов, а но-

мера транзакций возрастают последовательно). Но при вводе данных, относящихся к транзакциям типа AR и AP, возникают серьезные проблемы, поскольку нарушается последовательный порядок вставки. После перехода к выполнению, допустим, операции вставки данных по транзакции с ключом `AP000025`, в СУБД SQL Server обнаруживается, что на соответствующей странице недостаточно места, после чего СУБД находит в таблице строку с ключом `AR000001` и определяет, что операция вставки не является последовательной. Прежде чем появится возможность выполнить операцию вставки данных с ключом `AP000025`, необходимо скопировать половину строк со старой страницы на новую страницу.

Возникающие при этом издержки могут оказаться колоссальными. Напомним, что речь идет о кластеризованном индексе, а на нижнем уровне кластеризованного индекса находятся данные. Данные хранятся в последовательности, определяемой индексом. Это означает, что после перемещения строки индекса на новую страницу происходит также перемещение строки данных. А теперь предположим, что рассматриваемая бухгалтерская система эксплуатируется в типичной среде OLTP (трудно найти систему, в большей степени соответствующую определению среды OLTP, чем бухгалтерская система) и что огромный коллектив специалистов по обработке данных вводят в ней с максимально возможной скоростью счета-фактуры поставщиков или заказы заказчиков. А это означает, что постоянно происходят разбиения страниц, и при этом каждый раз пользователям таблицы приходится на время приостанавливать свою работу, пока система переносит данные с одной страницы на другую.

К счастью, как описано ниже, может быть предусмотрен ряд способов предотвращения возникновения подобной ситуации.

- Выбрать такой кластеризованный ключ, значения которого во время операций вставки будут возрастать последовательно. Для этого можно создать столбец идентификации, применить другой столбец, значения в котором возрастают последовательно при вводе данных о любой транзакции, независимо от системы, или, в условиях рассматриваемого примера, просто перенести обозначение типа в конец.
- Отказаться от определения кластеризованного индекса на таблице. Если имеет место ситуация, подобная описанной в данном примере, то указанный вариант часто является наилучшим, поскольку вставка данных с помощью некластеризованного индекса, заданного на неупорядоченной таблице, обычно происходит быстрее, чем вставка данных с помощью кластеризованного ключа.

Как уже было сказано, применение последовательно наращиваемых кластеризованных ключей позволяет уменьшить отрицательное влияние разбиения страниц, но необходимо также учитывать, что за это приходится платить. Одним из недостатков способа ввода данных с применением последовательно наращиваемых кластеризованных ключей является то, что этот способ не позволяет добиться высокой степени распараллеливания (при которой два или несколько пользователей пытаются получить доступ к одному и тому же объекту одновременно). Принимая решение о том, какой способ доступа к данным должен использоваться, необходимо учитывать, каковыми являются предъявляемые требования.

Приведенный выше пример, по-видимому, наиболее отчетливо демонстрирует причины, по которым автор стремится так подробно описать действия, выполняемые в базе данных в связи с применением индексов тех или других типов. Чтобы получить полное представление о том, при каких обстоятельствах индекс определенного типа становится наиболее подходящим (или неподходящим), необходимо

полностью разобраться в том, какие операции фактически выполняются в процессе обработки данных с помощью индексов.

Выбор правильного расположения столбцов в индексе

Наличие индекса, заданного на нескольких столбцах, не обеспечивает применения этого индекса для выполнения запросов, которые обращаются к отдельным столбцам, охваченным индексом.

Индекс рассматривается как применимый, только если в запросе используется первый столбец, перечисленный в индексе. Преимуществом указанного условия применения индексов является то, что не нужно добиваться точного взаимно однозначного согласования всех столбцов, указанных в запросе и охваченных индексом. Безусловно, чем больше количество последовательно согласованных столбцов в запросе и индексе (начиная от первых), тем лучше, но полный отказ от использования индекса происходит только в том случае, если в запросе не указан первый столбец индекса.

Указанный подход является вполне оправданным. Например, в телефонном справочнике данные об абонентах отсортированы в первую очередь по фамилиям, а затем по именам. Разве можно найти в таком справочнике абонента, если известно только имя, скажем, Фрэд? С другой стороны, если известно лишь то, что абонент имеет фамилию Блейк, то такой справочник поможет по крайней мере сузить область поиска.

Одна из наиболее распространенных ошибок в создании индексов, которая обнаруживается на практике, обусловлена тем, что, по мнению многих разработчиков, достаточно включить все столбцы в один индекс и использовать его во всех ситуациях. Но в действительности применение такого подхода приводит лишь к дублированию хранимых данных. Если в конструкции `JOIN`, `ORDER BY` или `WHERE` запроса не упоминается первый столбец индекса, то индекс полностью игнорируется.

Удаление индексов

В процессе эксплуатации базы данных необходимо время от времени проверять, какие запросы выполняются в ней чаще всего, и в случае необходимости вводить дополнительные индексы. Но не следует также забывать о том, что иногда бывает целесообразно удалить некоторые индексы. Необходимо помнить о том, что для сопровождения индексов при выполнении операций вставки требуются определенные издержки. Это означает, что нельзя руководствоваться стремлением вводить в действие все новые и новые необходимые для работы индексы, но не стремиться при этом удалять такие индексы, необходимость в использовании которых уже отпала. Следует всегда проверять, не нужно ли избавиться от каких-либо индексов.

Для удаления индекса применяется в основном такой же синтаксис, как и для удаления таблицы. Единственное отличие состоит в том, что имя удаляемого индекса должно быть уточнено путем указания имени таблицы (или представления), на которой он определен:

```
DROP INDEX <table or view name>.<index name>
```

После выполнения оператора удаления индекс становится недоступным.

Использование программы-мастера Index Tuning Wizard

Программисты, которые достаточно полно изучили все сведения, касающиеся использования индексов, фактически не ощущают потребности в применении программы **Index Tuning Wizard**, но эта программа все равно может оказаться весьма полезной. Функционирование программы Index Tuning Wizard основано на том, что берется файл регистрации рабочей нагрузки, созданный с использованием программы SQL Server Profiler (которая будет описана в главе 19), и осуществляется поиск информации, на основании которой может быть принято решение о том, какие индексы являются наиболее приемлемыми в конкретной системе.

Программа Index Tuning Wizard входит в состав инструментальных средств, доступ к которым предоставляется с помощью меню Tools программы SQL Server Management Studio. Кроме того, программа Index Tuning Wizard может быть вызвана на выполнение с помощью отдельного элемента меню программ в меню `Start` операционной системы Windows. Как и в отношении большинства других инструментальных средств настройки, автор не рекомендует использовать это инструментальное средство в качестве единственного способа принятия решения о том, какие индексы должны быть созданы, но эта программа может оказаться весьма удобной с той точки зрения, что некоторые предлагаемые ею рекомендации могут просто оказаться не вполне очевидными.

Сопровождение индексов

Разработчики часто грешат тем, что полностью забывают о существовании выпущенного ими программного продукта после передачи заказчику. Применительно к программному обеспечению многих типов такое поведение вполне оправдано — работа закончена, поставка выполнена, можно переходить к разработке следующего программного продукта или приступать к подготовке очередного выпуска программы. Тем не менее от дальнейшего сопровождения проектов, основанных на использовании базы данных, избавиться практически невозможно. Разработчик продолжает нести ответственность за исправное функционирование созданного им программного продукта даже по истечении даты доставки.

Сказанное выше вовсе не означает, что автор предлагает разработчику занять какую-то должность в отделе технического сопровождения заказчика. Речь идет о чем-то гораздо более важном — о **планировании сопровождения**.

При сопровождении индексов приходится решать две проблемы:

- устранять последствия разбиения страниц;
- устранять фрагментацию.

Влияние этих двух проблем на функционирование базы данных выражается в том, что снижается плотность заполнения страниц индекса. Безусловно, симптомы этих двух нарушений существенно различаются, но инструментальные средства устранения этих нарушений в работе являются одинаковыми, как и методы их устранения.

Фрагментация

Значительная часть настоящей главы была посвящена описанию того, как происходит разбиение страниц, но фактически еще ничего не было сказано по поводу

возникающей при этом фрагментации. Речь не идет о тех общеизвестных проблемах фрагментации файлов операционной системы, для устранения которых предусмотрен целый ряд инструментальных средств дефрагментации. К сожалению, подобные программные средства не могут помочь устранить фрагментацию базы данных.

Фрагментация базы данных возникает в результате того, что объем базы данных возрастает, происходит разбиение страниц, а затем в конечном итоге часть данных удаляется. Безусловно, механизм сопровождения B-дерева неплохо справляется с задачей обеспечения сбалансированности индекса при выполнении операций вставки данных, но практически не позволяет достичь той же цели, когда удаляются данные. В результате может сложиться такая ситуация, что на отдельных страницах будут находиться всего лишь одна-две строки. Иными словами, многие страницы будут содержать небольшое количество данных, составляющее лишь незначительную долю от того количества данных, которые они могли бы содержать.

Фрагментация становится причиной возникновения целого ряда проблем. Первая проблема является вполне очевидной – непроизводительное расходование внешней памяти. Напомним, что в СУБД SQL Server распределение пространства осуществляется путем выделения сразу целого экстента. Даже если требуется записать в память только одну страницу с единственной строкой, распределяется целый экстент.

Вторая проблема в большей степени касается снижения производительности операций доступа к данным. Дело в том, что если база данных фрагментирована, то строки данных в ней не сосредоточены в одном месте, а распределены по всему пространству памяти, поэтому в процессе выборки данных возникают дополнительные издержки. Скажем, вместо загрузки одной страницы и получения всех искомых десяти строк в СУБД SQL Server для получения той же информации приходится загружать десять отдельных страниц, поскольку доступ к строкам предоставляется только после считывания страницы, на которой они находятся. А чем больше страниц приходится считывать для получения одних и тех же данных, тем больший объем работы приходится при этом выполнять.

Тем не менее фрагментация базы данных имеет и свою положительную сторону, поскольку в результате фрагментации в системе OLTP повышается производительность выполнения операций вставки данных. Дело в том, что в результате разбиения страниц количество данных, хранимых на каждой отдельной странице, уменьшается, поэтому вставка данных на страницы, подвергшиеся разбиению, происходит быстрее.

Таким образом, увеличение степени фрагментации приводит к уменьшению производительности операций выборки данных, но вместе с тем способствует значительному повышению производительности операций вставки данных. Из этого следует, что для систем OLAP фрагментация является неблагоприятным фактором, а в системах OLTP играет двоякую роль.

Получение сведений о фрагментации и оценка вероятности разбиения страниц

Программное обеспечение SQL Server предоставляет возможность определить, какова степень заполнения страниц и экстентов в базе данных. На следующем этапе полученная информация может использоваться для принятия определенных решений по сопровождению базы данных. Для этой цели применяется команда `SHOWCONTIG`, которая фактически является опцией вызова на выполнение модуля DBCC (Database Consistency Checker – модуль контроля непротиворечивости базы данных).

Синтаксис оператора вызова на выполнение команды SHOWCONTIG довольно прост:

```
DBCC SHOWCONTIG
   [({<table name>|<table id>|<view name>|<view id>}
     [, <index name>|<index id>])]
   [WITH {ALL_INDEXES|FAST [, ALL_INDEXES ]|TABLERESULTS [, ALL_INDEXES]}]
     [ , { FAST | ALL_LEVELS } ]
DBCC SHOWCONTIG ([<table object id>], [<index id>])
```

В качестве примера ниже показано, как получить информацию об индексе PK_Order_Details, заданном на таблице Order Details.

```
USE Northwind
GO
DBCC SHOWCONTIG (@id, @IdxID)
GO
```

Очевидно, что полученные выходные данные требуют определенных пояснений:

```
DBCC SHOWCONTIG scanning 'Order Details' table...
Table: 'Order Details' (325576198); index ID: 1, database ID: 6
TABLE level scan performed.
Pages Scanned.....................................: 9
Extents Scanned...................................:  6
Extent Switches...................................: 5
Avg. Pages per Extent.............................:   1.5
Scan Density [Best Count:Actual Count]........:      33.33% [2:6]
Logical Scan Fragmentation ......................:    0.00%
Extent Scan Fragmentation .......................:    16.67%
Avg. Bytes Free per Page.........................:    673.2
Avg. Page Density (full).........................:  91.68%
DBCC execution completed. If DBCC printed error messages, contact your
system administrator.
```

Описание статистических показателей, применяемых в выводе оператора DBCC SHOWCONTIG, приведено в табл. 9.2.

Информация, которая формируется с помощью оператора DBCC SHOWCONTIG, может использоваться по-разному, в зависимости от конкретных условий.

В частности, вывод команды SHOWCONTIG позволяет получить значительный объем сведений о том, являются ли страницы и экстенты базы данных полностью заполненными, в значительной степени фрагментированными, или находятся в каком-то промежуточном состоянии (последний вариант, по-видимому, является наиболее вероятным). Если база данных применяется в системе OLAP, то при высокой плотности заполнения страниц быстродействие системы повышается, а при наличии значительной степени фрагментации система функционирует менее эффективно. В системе OLTP проявляются почти противоположные закономерности (хотя и до определенной степени).

Для того чтобы перейти к описанию того, какие действия следует предпринимать, если состояние базы данных является неудовлетворительным, необходимо вначале рассмотреть такие понятия, как перестройка индекса и коэффициент заполнения.

Таблица 9.2. Статистические показатели, применяемые в выводе оператора DBCC SHOWCONTIG

Статистический показатель	Описание
Pages Scanned	Количество страниц в таблице (на которой задан кластеризованный индекс) или в индексе
Extents Scanned	Количество экстентов в таблице или индексе. Минимальное значение этого показателя равняется количеству страниц, деленному на 8, а затем округленному в большую сторону. Чем больше количество экстентов в расчете на одно и то же количество страниц, тем выше фрагментация
Extent Switches	Количество операций перехода от одного экстента к другому, выполненных в модуле DBCC в ходе перебора страниц таблицы или индекса. Это — еще один показатель, позволяющий оценивать степень фрагментации: чем больше количество переходов, которые пришлось выполнить для просмотра одного и того же количества страниц, тем выше степень фрагментации
Avg. Pages per Extent	Среднее количество распределенных страниц в расчете на один экстент. При полном заполнении экстентов этот показатель имеет значение 8
Scan Density [Best Count: Actual Count]	Показатель Best Count определяет идеальное количество переходов с одного экстента на другой при условии безукоризненно качественного расположения данных в таблице. Показатель Actual Count показывает фактическое количество переходов с одного экстента на другой. Показатель Scan Density позволяет узнать плотность полного просмотра — процентное значение, полученное путем деления идеального количества переходов на фактическое количество переходов
Logical Scan Fragmentation	Процентная доля смещенных страниц (страниц, не находящихся на предназначенном для них месте в заданной последовательности), которая определяется путем просмотра страниц листового уровня индекса. Данный показатель характеризует только результаты операций полного просмотра, относящиеся к кластеризованной таблице. Как смещенная рассматривается такая страница, для которой следующая страница, указанная в схеме распределения индекса (Index Allocation Map — IAM), отличается от той, на которую направлен указатель следующей страницы на странице листового узла
Extent Scan Fragmentation	Этот показатель позволяет определить, имеются ли какие-либо экстенты, не находящиеся физически рядом с тем экстентом, с которым они должны находиться рядом логически. Показатель Extent Scan Fragmentation дает возможность узнать, не нарушена ли физическая последовательность страниц листового уровня индекса (хотя при этом логическая последовательность может оставаться правильной), а также выяснить, какая процентная доля экстентов хранится с нарушением заданного порядка следования
Avg. Bytes free per page	Среднее количество свободных байтов на страницах, считанных в результате полного просмотра. Этот показатель может принимать неоправданно высокое значение, если используемые строки имеют большую длину. Например, если строки имеют длину 4040 байтов, то на каждой странице помещается только одна строка, а среднее количество свободных байтов всегда составляет приблизительно 4020 байтов. При этом создается впечатление, что среднее количество свободных байтов весьма велико, но при такой величине строки фактически не может быть достигнуто более низкое значение данного показателя
Avg. Page density (full)	Средняя плотность страницы (в процентах). При вычислении значения этого показателя учитывается размер строки, поэтому он может служить более точным индикатором того, насколько полно заполнены страницы. Чем выше полученное процентное значение, тем лучше

Команда DBREINDEX и опция FILLFACTOR

Как уже было сказано выше, программное обеспечение SQL Server поддерживает опцию, позволяющую управлять тем, насколько высокой должна быть степень заполнения страниц листового уровня, а также, при желании, еще одну опцию, от которой зависят показатели заполнения страниц нелистового уровня. К сожалению, эти две опции являются превентивными, поскольку применяются лишь во время создания индекса. Поэтому в процессе эксплуатации базы данных приходится повторно применять опции, регламентирующие степень заполнения, выполняя перестройку индекса и снова вводя в действие указанные опции.

Для перестройки индексов применяются два способа: уничтожение и повторное создание индекса (при использовании этого способа настоятельно рекомендуется задавать в операторе создания индекса опцию DROP_EXISTING) или восстановление индекса с помощью команды DBREINDEX. Команда DBREINDEX также относится к категории команд DBCC и имеет такую синтаксическую структуру:

```
DBCC DBREINDEX (<'database.owner.table_name'>[, <index name>
[, <fillfactor>]]) [WITH NO_INFOMSGS]
```

После вызова этой команды на выполнение происходит полное восстановление требуемого индекса. Если указано только имя таблицы без имени индекса, <index name>, то восстанавливаются все индексы для требуемой таблицы. Какая-либо отдельно взятая команда, предназначенная для восстановления всех индексов в базе данных, не предусмотрена.

В результате восстановления индексов с применением оператора DBCC DBREINDEX происходит реструктуризация всей информации, содержащейся в индексах, а также вновь устанавливаются базовые показатели заполнения страниц. Если рассматриваемый индекс является кластеризованным, то происходит также реорганизация физических данных.

По умолчанию воссоздание страниц индекса происходит с учетом того условия, чтобы на каждой странице находилось максимальное количество строк за вычетом двух. Так же как и при использовании оператора CREATE TABLE, во время вызова на выполнение оператора DBCC DBREINDEX может быть задана опция <fillfactor>, аналогичная FILLFACTOR, которая может принимать любое значение от 0 до 100. Это число определяет процент заполнения страниц, вернее, то значение, которое будет характеризовать степень заполнения страниц после завершения перестройки индексов. Напомним, что после каждого разбиения страницы данные по-прежнему распределяются между двумя страницами поровну; таким образом, единственный способ постоянного поддержания заданного значения процента заполнения состоит в осуществлении регулярного восстановления индексов.

> Если в качестве значения процента заполнения будет указан нуль, то заполнение страниц происходит не с учетом заданного процента заполнения, а по другому принципу. А именно, на страницу записывается максимально возможное количество строк за вычетом двух (об этом следует помнить, выбирая значение опции <fillfactor>).

Как уже было сказано, значение опции <fillfactor> (соответственно FILLFACTOR) используется в тех случаях, когда возникает необходимость откорректировать плотность заполнения страниц. Кроме того, более низкие значения плотности заполнения

страницы (и поэтому низкие значения FILLFACTOR) идеально подходят для систем OLTP, в которых осуществляется большое количество операций вставки, поскольку это способствует предотвращению разбиений страниц. А более высокие значения плотности страниц применимы для систем OLAP (уменьшается количество считываемых страниц, и вместе с тем почти отсутствует вероятность того, что в базе данных будет происходить разбиение страниц, поскольку количество операций вставки невелико или такие операции вообще не выполняются).

Рассмотрим пример восстановления индекса, определенного на первичном ключе таблицы Order Details, который был создан ранее в этой главе со значением коэффициента заполнения, равным 65. Для можно воспользоваться следующей командой DBCC:

```
DBCC DBREINDEX ([Order Details], PK_Order_Details, 65)
```

После этого снова вызовем на выполнение оператор DBCC SHOWCONTIG, чтобы ознакомиться с полученными результатами:

```
DBCC SHOWCONTIG scanning 'Order Details' table...
Table: 'Order Details' (325576198); index ID: 1, database ID: 6
TABLE level scan performed.
Pages Scanned.....................................:   13
Extents Scanned...................................:   2
Extent Switches...................................:   1
Avg. Pages per Extent.............................:   6.5
Scan Density [Best Count:Actual Count]........:   100.00% [2:2]
Logical Scan Fragmentation .......................:   0.00%
Extent Scan Fragmentation ........................:   50.00%
Avg. Bytes Free per Page..........................:   2957.2
Avg. Page Density (full)..........................:   63.46%
DBCC execution completed. If DBCC printed error messages, contact your
system administrator.
```

Наиболее существенное изменение коснулось значения показателя Avg. Page Density. Полученное число немного меньше 65%, поскольку в СУБД SQL Server приходится учитывать размеры страницы и строки, но достигнутая величина является наиболее близкой к требуемой из всех возможных.

При использовании команды DBREINDEX и определении значения FILLFACTOR необходимо учитывать приведенные ниже рекомендации.

- Если значение опции <fillfactor> в команде DBREINDEX не задано, то при выполнении команды DBREINDEX используется значение этой опции, которое использовалось во время выполнения предыдущей операции создания индекса. Если значение опции <fillfactor> еще ни разу не было задано, то используется такой коэффициент заполнения, при котором происходит полное заполнение страницы за вычетом двух строк (но в большинстве ситуаций такое заполнение является слишком полным).
- Если значение опции <fillfactor> задано в команде DBREINDEX, то указанное значение становится применяемым по умолчанию значением опции FILLFACTOR для этого индекса.
- Безусловно, команда DBREINDEX может быть вызвана на выполнение во время работы базы данных в оперативном режиме, но автор настоятельно рекомендует не использовать такую возможность, поскольку в связи с восстановлени-

ем индекса блокируются ресурсы, а также может возникнуть целый ряд других проблем. По крайней мере, не следует выбирать для проведения такой операции период наибольшей нагрузки.

Резюме

Индексы играют чрезвычайно важную роль в организации функционирования SQL Server или любой другой среды баз данных, поэтому необходимо тщательно планировать их использование. Применение правильно выбранных индексов способствует повышению производительности, а при неправильном выборе индексов производительность может заметно снизиться.

Основные рекомендации по использованию индексов приведены ниже.

- Кластеризованные индексы обычно обеспечивают более высокое быстродействие, чем некластеризованные (следует учитывать, что при определенных условиях применение кластеризованных индексов приводит к снижению производительности).
- Некластеризованные индексы следует задавать только на таких столбцах, которые позволяют достичь высокого уровня избирательности (т.е. на столбцах, в которых 95% строк или больше являются уникальными).
- Применение индексов способствует ускоренному выполнению всех операторов языка манипулирования данными (Data Manipulation Language – DML), в том числе `INSERT`, `UPDATE`, `DELETE` и `SELECT`, но в целом в связи с необходимостью сопровождения индексов может происходить замедление при выполнении операций вставки, удаления и обновления (напомним, что операция обновления выполняется как две операции – удаления и вставки). Та часть процедуры выполнения запроса, которая связана с поиском, благодаря использованию индекса осуществляется быстрее, а другая часть запроса, при осуществлении которой происходит модификация данных, требует дополнительных издержек (в связи с необходимостью сопровождения индекса, а не только фактических данных).
- Индексы занимают определенную часть пространства в базе данных.
- Индексы используются, только если в условии запроса в первую очередь упоминается первый столбец индекса.
- Применение индексов может повлечь за собой не только повышение, но и снижение производительности, поэтому необходимо тщательно обосновывать необходимость создания каждого индекса и не формировать индексы, которые в действительности не требуются.
- Индексы XML позволяют достичь такой же производительности операций доступа к данным XML, имеющим сложную структуру, как и при обеспечении с помощью обычных индексов доступа к реляционным данным, характеризующимся гораздо более простой структурой, но следует учитывать, что применение индексов XML, как и всех прочих индексов, связано с возникновением дополнительных издержек.

В табл. 9.3 приведены контрольные вопросы, с помощью которых можно проще определить необходимые условия применения индексов.

Таблица 9.3. Контрольные вопросы, позволяющие определить условия применения индексов

Вопрос	Рекомендация
Много ли операций вставки или обновления применяется к рассматриваемой таблице?	Если ответ на этот вопрос является положительным, то сведите количество индексов к минимуму. В таблицах такого типа модификации обычно осуществляются путем осуществления поиска отдельных строки с помощью первичного ключа. Как правило, для такой таблицы достаточно предусмотреть лишь индекс, заданный на столбце (столбцах) первичного ключа. Если операции вставки осуществляются непоследовательно, то, по-видимому, целесообразно отказаться от использования кластеризованного индекса
Применяется ли данная таблица в основном для формирования отчетов? Иными словами, является ли таблица таковой, что в ней выполняется небольшое количество операций вставки, но на основе ее данных формируется большое количество различных отчетов?	Чем больше индексов, тем лучше. Задайте кластеризованный индекс на таком столбце (столбцах) с часто используемой информацией, в котором, по всей вероятности, выборка данных будет осуществляться в виде сплошных участков. В системах OLAP чаще всего применяется во много раз больше индексов, чем в системах OLTP
Являются ли данные таковыми, что определенный на них индекс характеризуется высокой степенью избирательности?	Если ответ на этот вопрос является положительным и столбец с этими данными часто упоминается в конструкциях `WHERE`, то задайте на нем индекс.
Уничтожены ли все индексы, которые больше не требуются?	Если ответ на этот вопрос является отрицательным, почему это не сделано?
Предусмотрена ли утвержденная стратегия сопровождения индексов?	Если ответ на этот вопрос является отрицательным, почему это не сделано?

Упражнения

9.1. Назовите по меньшей мере два способа определения индексов, заданных на таблице `HumanResources.Employee` в базе данных `AdventureWorks`.

9.2. Создайте некластеризованный индекс на столбце `ModifiedDate` таблицы `Production.ProductModel` в базе данных `AdventureWorks`.

9.3. Удалите индекс, созданный при выполнении упр. 9.2.

10

Представления

В предыдущих главах в основном рассматривались реальные объекты базы данных (т.е. объекты, существующие в определенном смысле самостоятельно), а в этой главе, начиная с представлений, мы переходим к описанию виртуальных объектов базы данных (которые также можно считать "виртуальными" лишь в определенном смысле).

Практика показывает, что представления используются либо слишком часто, либо слишком редко, иными словами, в действительности почти не встречаются такие проекты, в которых степень применения представлений была бы вполне оправдана. Изучение этой главы позволит читателю приобрести способность использовать представления для достижения описанных ниже целей.

- Сократить кажущуюся сложность базы данных для конечных пользователей.
- Обеспечить доступ пользователей ко всем необходимым данным и вместе с тем исключить возможность выборки конфиденциальной информации, хранимой в некоторых столбцах.
- Предусмотреть в базе данных дополнительные средства индексации, позволяющие повысить производительность запросов даже в том случае, если представления, на которых основаны индексы, не предназначены для использования по другому назначению.

В действительности представление по самой своей сути является не чем иным, как хранимым запросом. Весьма удобное свойство представлений состоит в том, что они позволяют объединять и согласовывать данные из базовых таблиц (или других представлений) для создания новых представлений, которые в большинстве отношений почти ничем не отличаются от базовых таблиц. В основе представления может лежать простой запрос, с помощью которого осуществляется выборка части столбцов только из одной таблицы, а остальные остаются нетронутыми, или сложный запрос, в котором соединяется несколько таблиц и создается один объект, полностью аналогичный отдельной таблице.

Простые представления

Синтаксическая структура оператора создания представления (в его наиболее простой форме) может рассматриваться как комбинация тех конструкций, которые уже были описаны в этой книге, – простого оператора CREATE, описанного в главе 5, а также оператора SELECT, неоднократно использовавшегося на протяжении данной книги:

```
CREATE VIEW <view name>
AS
<SELECT statement>
```

Безусловно, синтаксическая структура приведенного выше оператора сведена к минимуму, но в большинстве ситуаций эта структура отвечает всем потребностям в создании представлений. А в более расширенной форме синтаксическая структура оператора создания представления выглядит следующим образом:

```
CREATE VIEW [schema_name].<view name> [(<column name list>)]
[WITH [ENCRYPTION] [, SCHEMABINDING] [, VIEW_METADATA]]
AS
<SELECT statement>
WITH CHECK OPTION
```

Ниже в этой главе каждый компонент этого оператора будет рассматриваться отдельно, но вначале рассмотрим конкретный пример чрезвычайно простого представления.

Практическое занятие Создание простого представления

Создадим в базе данных Accounting представление CustomerPhoneList_vw, с помощью которого будет формироваться список номеров телефонов заказчиков:

```
USE Accounting
GO
CREATE VIEW CustomerPhoneList_vw
AS
   SELECT CustomerName, Contact, Phone
   FROM Customers
```

Следует отметить, что после вызова на выполнение этого оператора CREATE в программе Management Studio формируются такие же результаты, как и при использовании всех прочих операторов CREATE. А именно, не происходит возврат каких-либо строк, а вырабатывается только сообщение, позволяющее убедиться в успешном создании представления:

```
Command(s) completed successfully.
```

Теперь перейдите к использованию отображения в виде сетки (Results In Grid), если вы еще этого не сделали, чтобы было проще рассматривать результирующие наборы, если их несколько. Затем выполните один оператор SELECT применительно к созданному представлению (для этого используется точно такой же формат, как и для таблицы) и еще один – непосредственно к таблице Customers:

```
SELECT * FROM CustomerPhoneList_vw

SELECT * FROM Customers
```

Описание полученных результатов

Два полученных результирующих набора должны быть почти идентичными, а в действительности, что касается общих столбцов в представлении и таблице, эти два результирующих набора должны совпадать. Чтобы было проще понять, как запрос в СУБД SQL Server преобразуется в представление, рассмотрим поэтапно осуществляемые при этом действия.

Итак, оператор SELECT в представлении определен следующим образом:

```
SELECT CustomerName, Contact, Phone
FROM Customers
```

Поэтому после вызова на выполнение приведенного ниже оператора в СУБД SQL Server по существу передается команда: "Выполнить выборку всех данных из результирующего набора, полученного после выполнения оператора SELECT CustomerName, Contact, Phone FROM Customers".

```
SELECT * FROM CustomerPhoneList_vw
```

Таким образом, с помощью представления осуществляется своего рода промежуточная обработка. Иными словами, представление фактически не изменяет какие-либо исходные данные, а скорее лишь "собирает в себе" выбранную по условию версию данных, к которым в нем осуществляется доступ. Этим свойством представлений можно пользоваться для создания менее сложного образа базы данных, предстающего перед конечным пользователем. Безусловно, в базе данных предусмотрено уже очень много инструментальных средств, позволяющих упростить работу пользователя, поэтому на первый взгляд может показаться, что представления не являются в этом отношении чем-то особенным, но пользователи так не считают.

Однако следует учитывать, что по умолчанию в СУБД SQL Server не предусматривается выполнение каких-либо особых действий применительно к представлениям. Запрос, лежащий в основе представления, вызывается на выполнение точно так же, как если бы это был запрос, введенный в командной строке, т.е. какая-либо предварительная оптимизация не предусматривается. Это означает, что после создания представления вводится еще один этап обработки на пути от формирования запроса к данным до получения данных, т.е. затраты на обработку данных увеличиваются. Иными словами, с помощью представлений никогда не удастся добиться такого же быстродействия, как при непосредственном вызове на выполнение оператора SELECT, лежащего в основе этого представления. Но несмотря на сказанное, в связи с необходимостью обеспечения защиты данных или упрощения работы пользователей применение представлений часто бывает вполне обоснованным. Следует лишь учитывать, оправданы ли дополнительные издержки с точки зрения преимуществ, достигаемых в конкретной ситуации.

Рассмотрим еще одни пример представления, который показывает возможности использования представлений для сокрытия конфиденциальных данных. Для этого снова воспользуемся таблицей Employees базы данных Accounting. Эта таблица состоит из столбцов, перечисленных в табл. 10.1.

Таблица 10.1. Столбцы таблицы Employees

Столбцы таблицы Employees
EmployeeID
FirstName
MiddleInitial
LastName
Title
SSN
Salary
HireDate
TerminationDate
ManagerEmpID
Department

Определенная часть информации, содержащейся в таблице Employees, согласно федеральному закону США, должна быть скрыта от постороннего взгляда (доступ к этой информации должен предоставляться только с учетом действительных потребностей), а другие столбцы таблицы содержат информацию, доступ к которой может быть предоставлен любому желающему. Предположим, что требуется предоставить группе пользователей неограниченные возможности выборки данных из столбцов, допускающих свободный доступ, но исключить для этих пользователей возможность видеть общую структуру таблицы или обращаться ко всем данным. Одним из решений могло бы стать сопровождение отдельной таблицы, которая включает только необходимые столбцы, как показано в табл. 10.2.

Таблица 10.2. Общедоступные столбцы таблицы Employees

Общедоступные столбцы таблицы Employees
EmployeeID
FirstName
MiddleInitial
LastName
Title
HireDate
TerminationDate
ManagerEmpID
Department

Безусловно, на первый взгляд создается впечатление, что такое решение соответствует нашим потребностям, однако при его осуществлении возникают описанные ниже серьезные проблемы.

- Объем используемого дискового пространства увеличивается вдвое.
- Возникает проблема синхронизации, обусловленная тем, что изменения, внесенные в одну таблицу, должны быть отражены в другой.
- Количество выполняемых операций ввода-вывода также увеличивается вдвое (операции чтения и записи данных применяются к двум таблицам, а не к одной) при каждой вставке, обновлении или удалении строк.

Простым и довольно изящным решением указанной задачи может стать представление. Если используется представление, то не требуется создавать копии данных, т.е. в базе данных хранится только один экземпляр данных (в таблице или таблицах, лежащих в основе представления), поэтому устраняются все недостатки, которые были описаны выше. Вместо создания и сопровождения полностью отдельной таблицы можно просто создать представление, которое по своим функциональным возможностям почти ничем не отличается от отдельной копии таблицы.

В настоящее время созданная нами таблица `Employees` пуста. Чтобы ввести в нее некоторые строки, загрузите файл `Chapter10.sql` (предоставляемый в составе исходного кода) в программу Management Studio и вызовите его на выполнение. Затем введите следующее определение представления в базу данных `Accounting`:

```
USE Accounting
GO
CREATE VIEW Employees_vw
AS
SELECT  EmployeeID,
        FirstName,
        MiddleInitial,
        LastName,
        Title,
        HireDate,
        TerminationDate,
        ManagerEmpID,
        Department
FROM Employees
```

После этого появляется возможность предоставить доступ к соответствующим данным таблицы `Employees` любым пользователям (прямо или косвенно). Это означает, что обеспечивается возможность предоставлять полный доступ к таблице `Employees` только тем пользователям, которым действительно требуется эта информация, но запретить непосредственный доступ к этой таблице всем прочим пользователям. Вместо этого пользователям, в служебные обязанности которых не входит ознакомление со всей информацией таблицы `Employees`, может быть предоставлен доступ к представлению `Employees_vw`. Все желающие воспользоваться этой информацией смогут обратиться к ней так же, как и при использовании обычной таблицы:

```
SELECT *
FROM Employees_vw
```

Рассматриваемый пример фактически еще раз подчеркивает необходимость принятия и соблюдения определенных соглашений об именовании, о чем уже неоднократно было сказано в настоящей книге. В данном случае автор использует для обозначения представления суффикс `_vw`, и это позволяет легко определить, что данное имя относится к представлению, а не к таблице. Но иногда от пользователя приходится скрывать некоторые детали организации средств доступа к базе данных, поэтому специально убирать обозначение `_vw` в именах некоторых представлений. Если бы такое решение было принято в данном случае, то пришлось бы назвать представление по-другому (поскольку имя `Employees` уже закреплено за базовой таблицей), но в практической работе неоднократно приходится убеждаться в том, насколько редко встречаются пользователи, способные уловить различие между представлением и таблицей, если нельзя определить по имени, что это — представление, а не таблица.

Использование представлений как средств выборки по условию

По-видимому, данный раздел — один из самых коротких в книге, поскольку трудно найти более простую тему по сравнению с изложенной в нем.

Выше в данной главе уже было сказано, как создать простое представление, — для этого достаточно включить в оператор создания представления конструкцию `SELECT`. А для обеспечения выборки данных в представлении по условию, как и в случае запроса, достаточно включить конструкцию `WHERE`.

Еще раз вернемся к примеру представления `Employees_vw`, который рассматривался в предыдущем разделе, и внесем в него небольшое дополнение для того, чтобы с его помощью формировался только список служащих, которые в настоящее время входят в кадровый состав. Для этого необходимо внести всего лишь два изменения.

Прежде всего необходимо исключить по условию из состава результатов данные о тех служащих, которые больше не работают в компании. В качестве признака того, что служащий в настоящее время не работает в компании, будем рассматривать наличие в данных об этом служащем информации о дате увольнения. Очевидно, что если служащий работает в компании, то в поле `TerminationDate` записи с данными об этом служащем содержится NULL-значение; на этом может быть основан критерий выборки в запросе.

Второе изменение иллюстрирует еще одну простую особенность представлений, используемых исключительно в качестве запросов. Дело в том, что в таких представлениях столбец (столбцы), содержавшийся в конструкции `WHERE`, не обязательно должен быть включен в список выборки. А в данном случае нет никакого смысла включать информацию о дате увольнения в результирующий набор, поскольку нас интересуют только данные о служащих, которые в настоящее время продолжают работать в компании.

Практическое занятие

Использование представлений для выборки данных по условию

С учетом двух описанных выше особенностей представлений создадим новое представление на основе старого, внеся в него небольшие изменения:

```
CREATE VIEW CurrentEmployees_vw
AS
SELECT   EmployeeID,
         FirstName,
         MiddleInitial,
         LastName,
         Title,
         HireDate,
         ManagerEmpID,
         Department
FROM Employees
WHERE TerminationDate IS NULL
```

В этом примере заслуживает внимания то, что изменилось имя представления и дополнительно введена конструкция WHERE; кроме того, из списка выборки удален столбец TerminationDate.

Проверим работу этого представления путем вызова на выполнение несложного оператора SELECT применительно к таблице Employees и указания в списке выборки только тех данных, которые нас интересуют:

```
SELECT   EmployeeID,
         FirstName,
         LastName,
         TerminationDate
FROM Employees
```

В результате из всего объема данных таблицы будет получено лишь несколько столбцов:

```
EmployeeID       FirstName        LastName       TerminationDate
-----------      -------------    ------------   -------------------------
1                Joe              Dokey          NULL
2                Peter            Principle      NULL
3                Steve            Smith          1997-01-31 00:00:00
4                Howard           Kilroy         NULL
5                Mary             Contrary       1998-06-15 00:00:00
6                Billy            Bob            NULL

(6 row(s) affected)
```

Теперь вызовем на выполнение рассматриваемое представление:

```
SELECT   EmployeeID,
         FirstName,
         LastName
FROM CurrentEmployees_vw
```

Полученный при этом результирующий набор становится немного меньше:

```
EmployeeID       FirstName        LastName
------------     ----------       ---------------
1                Joe              Dokey
2                Peter            Principle
4                Howard           Kilroy
6                Billy            Bob

(4 row(s) affected)
```

В этих данных отсутствуют сведения о некоторых служащих, а этого и требовалось добиться при создании представления.

Описание полученных результатов

Как уже было сказано выше, рассматриваемое представление фактически является не чем иным, как оператором SELECT, который скрыт от пользователей. Таким образом, пользователи получают возможность отвлечься от того, что содержится в самом операторе SELECT, и учитывать лишь результаты, сформированные с его помощью, на таком же основании, как если бы эти результаты представляли собой отдельную таблицу. Такую ситуацию можно сравнить с ситуацией применения производных таблиц, которые были описаны в главе 7. Кроме того, после вызова представления путем указания его имени осуществляется выборка данных по условию, поэтому в запросе, в котором указано имя этого представления, даже не приходится регламентировать условия выборки данных (поскольку это уже сделано с помощью представления).

Более сложные представления

Представления, рассматриваемые в этом разделе, действительно являются более сложными по сравнению с описанными выше, но вполне доступны для освоения. Чаще всего наиболее сложные понятия, с которыми приходится сталкиваться при изучении представлений, все равно гораздо проще, чем многие другие концепции языка SQL.

В действительности сложные представления отличаются от простых лишь тем, что в них дополнительно используются соединения, операции агрегирования, а также, возможно, некоторые варианты переименования столбцов.

По-видимому, представления используются чаще всего для упрощения структуры данных, иными словами, для устранения тех сложностей доступа к данным, которые были вкратце описаны в начале данной главы. Предположим, что необходимо подготовить представление для руководителей компании, чтобы им было проще контролировать данные о сбыте. Эту обязанность должен взять на себя программист, поскольку даже в наш информационный век редко встречаются менеджеры, которые сами способны написать сложные запросы (надеемся, что менеджеры, читая эту книгу, не воспримут данное утверждение как личное оскорбление).

В качестве примера снова рассмотрим базу данных Northwind. Предположим, что руководители компании хотели бы иметь возможность вводить простые запросы, позволяющие узнавать, какие заказы были размещены в компании, какие товары упоминаются в заказах и кто разместил эти заказы. Таким образом, необходимо создать представление, применительно к которому руководители смогут выполнять очень простые запросы (напомним, что это представление создается в базе данных Northwind):

```
USE Northwind
GO
CREATE VIEW CustomerOrders_vw
AS
SELECT   cu.CompanyName,
         o.OrderID,
         o.OrderDate,
```

```
        od.ProductID,
        p.ProductName,
        od.Quantity,
        od.UnitPrice,
        od.Quantity * od.UnitPrice AS ExtendedPrice
FROM    Customers AS cu
INNER JOIN     Orders AS o
     ON cu.CustomerID = o.CustomerID
INNER JOIN     [Order Details] AS od
     ON o.OrderID = od.OrderID
INNER JOIN     Products AS p
     ON od.ProductID = p.ProductID
```

Теперь выполним следующий простой оператор SELECT:

```
SELECT *
FROM CustomerOrders_vw
```

В результате этого можно обнаружить, что осуществляется вывод большого количества строк (свыше 2000), но вместе с тем оказывается, что полученная информация стала гораздо проще и доступнее для понимания и анализа со стороны среднего пользователя. Более того, даже без особой подготовки руководители компании (или любые заинтересованные пользователи) получают возможность непосредственно обращаться именно к тем данным, которые их интересуют, вводя, например, такие запросы:

```
SELECT CompanyName, ExtendedPrice
FROM CustomerOrders_vw
WHERE OrderDate = '9/3/1996'
```

Пользователи не обязаны знать, что данные, к которым они обращаются, получены в результате соединения четырех таблиц; все эти нюансы скрыты в представлении. Вместо этого пользователю достаточно применить лишь самые простейшие навыки (и в связи с этим не прилагать слишком значительных умственных усилий), чтобы получить все необходимые данные.

```
CompanyName             ExtendedPrice
--------------------    ------------------
LILA-Supermercado       201.6000
LILA-Supermercado       417.0000
LILA-Supermercado       432.0000

(3 row(s) affected)
```

Мало того, запрос, лежащий в основе представления, может стать еще более целенаправленным. Предположим, например, что с помощью представления достаточно лишь получить данные о сбыте за вчерашний день. Для этого достаточно ввести в запрос небольшие изменения, как показано ниже.

```
USE Northwind
GO

CREATE VIEW YesterdaysOrders_vw
AS
SELECT   cu.CompanyName,
         o.OrderID,
```

```
        o.OrderDate,
        od.ProductID,
        p.ProductName,
        od.Quantity,
        od.UnitPrice,
        od.Quantity * od.UnitPrice AS ExtendedPrice
FROM    Customers AS cu
INNER JOIN    Orders AS o
      ON cu.CustomerID = o.CustomerID
INNER JOIN    [Order Details] AS od
      ON o.OrderID = od.OrderID
INNER JOIN    Products AS p
      ON od.ProductID = p.ProductID
WHERE CONVERT(varchar(12),o.OrderDate,101) =
      CONVERT(varchar(12),DATEADD(day,-1,GETDATE()),101)
```

К сожалению, все даты, которые хранятся в базе данных Northwind, относятся к давно прошедшему времени, поэтому запрос к данному представлению вряд ли возвратит какие-либо данные, поэтому введем строку для проверки рассматриваемого представления. Вызовите один раз на выполнение следующий сценарий:

```
USE Northwind
DECLARE @Ident int
INSERT INTO Orders
(CustomerID,OrderDate)
VALUES
('ALFKI', DATEADD(day,-1,GETDATE()))
SELECT @Ident = @@IDENTITY
INSERT INTO [Order Details]
(OrderID, ProductID, UnitPrice, Quantity)
VALUES
(@Ident, 1, 50, 25)

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident)
```

В этом сценарии используются конструкции, которые будут более подробно описаны в главе, касающейся сценариев и пакетов. А на данный момент отметим, что этот сценарий позволяет ввести в базу данных Northwind такие значения, выборка которых должна быть осуществлена с помощью представления. После выполнения указанного сценария в программе Management Studio должен появиться результат, который выглядит примерно так:

```
(1 row(s) affected)
(1 row(s) affected)
------------------------------------------
The OrderID of the INSERTed row is 11087

(1 row(s) affected)
```

Следует учитывать, что некоторые из приведенных здесь сообщений появятся во вкладке **Messages**, *только если программа Management Studio эксплуатируется в режиме отображения результатов в сетке,* **Results In Grid**.

При выполнении этого примера на другом компьютере значение OrderID может измениться, но остальная часть результатов должна остаться в основном такой же.

Теперь выполним запрос применительно к рассматриваемому представлению, чтобы узнать, каковы будут результаты:

```
SELECT CompanyName, OrderID, OrderDate FROM YesterdaysOrders_vw
```

Полученные результаты действительно показывают, что в базе данных появилась информация о заказе с номером 11087 за вчерашний день:

```
CompanyName                    OrderID     OrderDate
------------------------------ ----------- -------------------------
Alfreds Futterkiste            11087       2000-08-05 17:37:52.520

(1 row(s) affected)
```

> Не следует придавать большого значения тому, что фактически полученные данные о номерах заказов, OrderID, будут отличаться от приведенных в книге. Дело в том, что эти номера присваиваются системой (поскольку OrderID — столбец идентификации) и зависят от того, сколько строк уже вставлено в таблицу. Иными словами, при выполнении данного примера читателем могут быть получены другие номера заказов.

Функции DATEADD и CONVERT

Безусловно, оператор создания представления с конструкцией JOIN, приведенный в предыдущем разделе, сложнее тех операторов, которые рассматривались до сих пор, но все еще остается вполне доступным для понимания. В нем одна за другой добавляются таблицы и столбцы новых таблиц применяются в операциях соединения с соответствующими им столбцами тех таблиц, которые были указаны ранее. Как всегда, следует отметить, что соединяемые столбцы не обязательно должны иметь одинаковые имени. Достаточно того, чтобы в этих столбцах находились данные, связанные друг с другом.

Но поскольку рассматриваемый оператор соединения является все же относительно сложным, проанализируем, какие действия осуществляются в запросе, который поддерживает это представление.

Наибольший интерес представляет конструкция WHERE этого оператора:

```
WHERE CONVERT(varchar(12),o.OrderDate,101) =
      CONVERT(varchar(12),DATEADD(day,-1,GETDATE()),101)
```

В данной конструкции применяется единственная операция сравнения, но для получения требуемого результата предусмотрено несколько функций.

На первый взгляд кажется, что достаточно лишь сравнить значение OrderDate в таблице Orders со значением GETDATE() (с нынешней датой) за вычетом одного дня (в данном случае функция DATEADD применяется исключительно для выполнения операции вычитания с датами). С помощью функции DATEADD можно складывать значения продолжительности любых промежутков времени (а вычитание выполняется, если используются отрицательные значения операндов этой функции). Достаточно только указать, к какой дате должна применяться операция сложения, с помощью какой единицы времени измеряется продолжительность суммируемого промежутка времени (сутки, недели, годы, минуты и т.д.), и вызвать на выполнение эту функцию.

Поэтому внешне такая задача выглядит таким образом, что достаточно просто получить значение сегодняшней даты с помощью функции GETDATE(), а затем вычесть одни сутки с помощью функции DATEADD. Но проблема состоит в том, что значение, полученное с помощью функции GETDATE(), включает текущее время суток, поэтому данному критерию сравнения будут соответствовать только те строки, относящиеся к предыдущей дате, которые содержат такое же значение времени суток с точностью до 3.3333 миллисекунд, а вероятность такого совпадения весьма мала. Поэтому в операторе сравнения предусмотрено дополнительное уточнение и используется функция CONVERT для исключения времени суток из обоих значений дат до выполнения операции сравнения. Таким образом, с помощью этого представления будет получена информация обо всех торговых сделках, происшедших в предыдущую дату в любое время.

Использование представлений для внесения изменений в данные до ввода в действие триггеров INSTEAD OF

Как уже было сказано, с точки зрения функциональных возможностей их использования представления почти полностью аналогичны таблицам (хотя, разумеется, для создания представлений применяются совсем другие конструкции по сравнению с таблицами). Но в этом разделе основное внимание уделено тому, в чем состоят функциональные различия между таблицами и представлениями.

Многие начинающие разработчики об этом не догадываются, но следует знать, что к представлениям могут успешно применяться операторы INSERT, UPDATE и DELETE. Тем не менее при использовании операций модификации данных с помощью представлений необходимо учитывать некоторые нюансы, описанные ниже.

- Если оператор выборки, лежащий в основе представления, содержит операцию соединения, то в большинстве случаев с помощью этого представления невозможно вставлять или удалять данные, применяя оператор INSERT или DELETE, если при этом не используется триггер INSTEAD OF. В некоторых случаях оператор UPDATE может применяться без триггера INSTEAD OF (например, при условии, что обновляются только те столбцы, которые относятся к одной и той же таблице), но требуется некоторая дополнительная подготовка, так как в противном случае при выполнении операторов обновления могут вскоре возникнуть проблемы.
- Если представление ссылается только на одну таблицу, то вставка данных с помощью оператора INSERT с использованием представления может осуществляться без применения триггера INSTEAD OF, при условии, что в представлении обеспечивается доступ ко всем обязательным (не допускающим неопределенных значений) столбцам таблицы или для этих столбцов заданы применяемые по умолчанию значения. Если же в таблице имеется столбец, который не определен в представлении и не имеет заданного по умолчанию значения, то при использовании триггера INSTEAD OF применение оператора INSERT допускается даже в случае представлений, относящихся к единственной таблице.
- Предусмотрена возможность, хотя и в ограниченной степени, регламентировать, какие данные могут и не могут быть вставлены или обновлены в представлении.

Как показывает приведенное описание условий модификации данных с помощью представлений, во многих случаях невозможно обойтись без триггеров INSTEAD OF. Но мы пока не можем продолжить описание этой темы, поскольку сами триггеры INSTEAD OF являются достаточно сложными объектами и к тому же в предыдущих главах данной книги еще не было приведено достаточно подробное описание триггеров. Автор уже неоднократно указывал, что при описании проблематики, связанной с СУБД SQL Server, часто приходится сталкиваться со старой дилеммой "курицы" и "яйца" (что появилось раньше?). Автор обязан раскрыть тему триггеров INSTEAD OF, поскольку она имеет непосредственное отношение к представлениям, но на страницах этой книги нельзя переходить к описанию триггеров INSTEAD OF, не изложив все необходимые сведения о тех объектах, на которых они создаются (речь идет о таблицах и представлениях).

Поэтому автор решил в этой главе раскрыть тематику представлений в том аспекте, в каком эти объекты применялись с самого начала, еще до введения таких конструкций, как триггеры INSTEAD OF. Но хотя при изучении настоящей главы читатель не будет знакомиться с конкретными сведениями о триггерах INSTEAD OF, он должен хорошо понимать, в чем состоит назначение этих триггеров. Таким образом, ознакомившись в данной главе с кратким обзором триггеров INSTEAD OF, мы снова вернемся к этой теме и раскроем ее более подробно в главе 15.

Но самая главная особенность триггеров INSTEAD OF должна быть отмечена в данной главе. Это — триггеры особого рода, которые по существу вызываются на выполнение "вместо" (instead of, отсюда их название) того оператора, который вызвал запуск триггера. Благодаря такой организации работы после запуска триггера можно определить, какие действия уже были выполнены с помощью оператора, а затем непосредственно в коде триггера принять решения, касающиеся того, как разрешить любые возникшие конфликты или устранить другие проблемы, которые, возможно, проявились к этому моменту. Таким образом, триггеры INSTEAD OF являются очень мощными, но вместе с тем довольно сложными, поэтому отложим их описание до одной из следующих глав.

Внесение изменений в данные с помощью представлений в условиях использования операций соединения

Если в операторе выборки, лежащем в основе представления, затрагивается больше чем одна таблица, то в большинстве случаев применение этого представления для модификации данных не допускается, если не предусмотрено использование триггера INSTEAD OF (как уже было сказано, дополнительная информация об этом будет приведена ниже). Корпорацией Microsoft было принято решение запретить по умолчанию модификацию данных с помощью представлений, в которых используется несколько таблиц, поскольку при этом возникают некоторые неоднозначности при выборе способов реализации операций внесения изменений. Но разработчик может взять на себя обязанности по устранению противоречий, используя триггер INSTEAD OF для анализа данных, подвергаемых модификации, и явно указывая СУБД SQL Server, какие действия необходимо выполнить с модифицируемыми данными.

Обязательное соблюдение требования об использовании в представлении всех столбцов, не допускающих неопределенных значений, или о применении заданных по умолчанию значений

По умолчанию, если для вставки данных используется представление, то необходимо предусмотреть возможность передачи в операторе вставки значений во все обязательные столбцы (столбцы, которые не допускают использования NULL-значений). Кроме того, само это представление должно содержать в основополагающем запросе оператор SELECT, относящийся к единственной таблице, или по крайней мере вставка данных с помощью представления должна ограничиваться воздействием только на одну таблицу, а также должны быть предусмотрены данные для всех обязательных столбцов. Следует отметить, что требование задавать значения для всех обязательных столбцов не подразумевает указания всех этих столбцов в списке выборки основополагающего оператора SELECT представления. Дело в том, что для предоставления значений, которые должны быть вставлены в столбцы, не допускающие использования неопределенных значений, вполне могут быть применены заданные по умолчанию значения. Таким образом, чтобы обеспечить вставку данных с помощью представления, выполняя операторы INSERT, достаточно обеспечить, чтобы этим представлением были охвачены все столбцы, для которых не предусмотрены заданные по умолчанию значения и которые не допускают наличия в них NULL-значений. Вполне очевидно, что единственный способ достичь этой цели в любых обстоятельствах состоит в применении триггера INSTEAD OF.

Проверка данных, вставляемых с помощью представления, — ключевое слово WITH CHECK OPTION

Многие средства СУБД SQL Server мало известны в широких кругах разработчиков, а возможности, предоставляемые конструкцией WITH CHECK OPTION, относятся к числу наименее известных. В основе применения этой конструкции лежит простое правило — обновление или вставка данных с помощью представления, в котором задана конструкция WITH CHECK OPTION, осуществляется, только если строка, полученная в результате применения этой операции обновления или вставки, соответствует критериям, допускающим ее включение в это представление. Иначе говоря, строка, полученная в результате вставки или обновления, должна соответствовать критериям конструкции WHERE, которая определена в операторе SELECT, лежащем в основе представления.

Практическое занятие Применение конструкции WITH CHECK OPTION

Чтобы рассмотреть, как используется конструкция WITH CHECK OPTION, продолжим работу с базой данных Northwind и с помощью приведенного ниже оператора создадим представление, в котором отображаются данные только о поставщиках из штата Орегон. Количество столбцов, которые могут рассматриваться как применимые для работы с таблицей Shippers, ограничено, поэтому для определения того, в каком штате находится поставщик, нам придется использовать коды городов (но вначале не забудьте указать, что используется база данных Northwind).

```
CREATE VIEW OregonShippers_vw
AS
```

```
SELECT   ShipperID,
         CompanyName,
         Phone
FROM     Shippers
WHERE Phone LIKE '(503)%'
   OR Phone LIKE '(541)%'
   OR Phone LIKE '(971)%'
WITH CHECK OPTION
```

После вызова на выполнение оператора SELECT * применительно к этому представлению обнаруживается, что происходит возврат всех строк таблицы (поскольку все строки в таблице соответствуют критериям, заданным в определении представления):

```
ShipperID        CompanyName                Phone
-----------      -------------------------  ----------------------
1                Speedy Express             (503) 555-9831
2                United Package             (503) 555-3199
3                Federal Shipping           (503) 555-9931
4                Speedy Shippers, Inc.      (503) 555-5566

(4 row(s) affected)
```

Теперь предпримем попытку обновить одну из строк с помощью этого представления — задать номер телефона, содержащий код города, отличный от любого из допустимых значений, т.е. начинающийся с подстроки, которая отличается от (503), (541) или (971):

```
UPDATE OregonShippers_vw
SET Phone = '(333) 555 9831'
WHERE ShipperID = 1
```

Попытка выполнить этот оператор немедленно приводит к формированию в СУБД SQL Server следующего сообщения об ошибке:

```
Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either
specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION
and one or more rows resulting from the operation did not qualify under the
CHECK OPTION constraint.
The statement has been terminated.
```

Описание полученных результатов

В конструкции WHERE предусмотрено условие выборки, согласно которому в представление входят только строки, содержащие в поле с номером телефона значения кодов городов (503), (541) или (971), а конструкция WITH CHECK OPTION указывает, что любые данные, вставляемые или обновляемые с помощью операторов INSERT или UPDATE, должны соответствовать критериям, заданным в конструкции WHERE (тогда как код города (333) этим критериям не соответствует).

Приведенный выше оператор обновления не соответствует критериям конструкции WHERE, поэтому отбрасывается. Но попытка вставить эти же данные непосредственно в базовую таблицу:

```
UPDATE Shippers
SET Phone = '(333) 555 9831'
WHERE ShipperID = 1
```

не рассматривается в СУБД SQL Server как неправильная, поэтому вырабатывается следующее сообщение:

```
(1 row(s) affected)
```

Таким образом, проверка, предусмотренная в конструкции WITH CHECK OPTION, применяется только к представлению, а не к основополагающей таблице. В определенных обстоятельствах возможность выбрать, использовать ли для модификации данных таблицу или представление, может действительно оказаться весьма удобной. Представьте себе такую ситуацию, когда требуется разрешить некоторым пользователям вставлять или обновлять данные в таблице, но только если обновляемые или вставляемые данные соответствуют определенным критериям. Безусловно, можно было бы легко реализовать подобное условие, предусмотрев использование ограничения CHECK на основополагающей таблице, но такое решение не всегда является идеальным.

Дело в том, что может потребоваться учесть еще одно такое пожелание, чтобы некоторые пользователи имели возможность вводить данные в таблицу с помощью операторов INSERT без учета указанных критериев. Но ограничение CHECK не позволяет предоставить разным пользователям различные возможности. А определив представление с конструкцией WITH CHECK OPTION, можно предоставить пользователям с ограниченными правами возможность применять для вставки данных только представление, а полноправным пользователям дать возможность выбирать — осуществлять ли вставку данных непосредственно в базовую таблицу или использовать представление, в тех случаях, если вставляемые данные соответствуют условиям, заданным в представлении.

Чтобы убедиться в том, что проверка данных, модифицируемых с помощью представления, распространяется и на операторы вставки, вызовем на выполнение оператор INSERT, который содержит данные, противоречащие условиям в конструкции WHERE представления:

```
INSERT INTO OregonShippers_vw
VALUES
('My Freight Inc.', '(555) 555-5555')
```

При осуществлении этой попытки, как и прежде, появляется уже знакомое сообщение об ошибке, в котором говорится о прекращении выполнения оператора:

```
Msg 550, Level 16, State 1, Line 1
The attempted insert or update failed because the target view either
specifies WITH CHECK OPTION or spans a view that specifies WITH CHECK OPTION
and one or more rows resulting from the operation did not qualify under the
CHECK OPTION constraint.
The statement has been terminated.
```

Редактирование представлений с помощью средств языка SQL

Для внесения изменений в определения представлений применяются два способа, основанные на использовании операторов языка SQL: модификация представления с помощью оператора ALTER VIEW или его уничтожение и повторное создание с помощью оператора CREATE VIEW. Но при использовании оператора ALTER VIEW следует помнить, что его выполнение также приводит к полной замене существующего представления. Различия между способами модификации определения представления, в которых используется оператор ALTER VIEW или CREATE VIEW, заключаются лишь в следующем:

- при выполнении оператора ALTER VIEW предполагается, что рассматриваемое представление уже существует, а при выполнении оператора CREATE VIEW такое предположение не делается;
- после выполнения оператора ALTER VIEW сохраняются все права на использование представления, предоставленные пользователям;
- после выполнения оператора ALTER VIEW сохраняется вся информация о зависимостях.

Наиболее важным является второе из указанных различий. Если не учитывать этого различия, то выполнение оператора DROP для уничтожения представления, а затем создание представления с помощью оператора CREATE приводит почти к таким же результатам, как и при использовании оператора ALTER VIEW. Но возникает важная проблема, связанная с тем, что приходится снова определять всю структуру прав, согласно которой СУБД принимает решение о том, кто может и не может применять это представление.

Уничтожение представления

Синтаксис оператора уничтожения представлений является чрезвычайно простым:

```
DROP VIEW <view name>, [<view name>,[ ...n]]
```

Создание и редактирование представлений в программе Management Studio

Некоторые разработчики не стремятся глубоко разобраться в том, чем они занимаются, поэтому охотно используют действительно превосходные возможности программы Management Studio. С помощью этой программы задача создания приложения значительно упрощается, причем для этого фактически не требуется глубоко разбираться в том, как действуют запросы.

Чтобы ознакомиться с этими средствами, запустите программу Management Studio, откройте подузел базы данных Northwind узла Databases и щелкните правой кнопкой мыши на обозначении Views. Появится окно, которое показано на рис. 10.1.

Рис. 10.1. Узел Databases

Теперь выберите команду New View, чтобы открыть новое диалоговое окно.

Это диалоговое окно позволяет выбирать таблицы, данные которых должны быть включены в представление. На рис. 10.2 показано, как выбрать одну таблицу, в данном случае `Categories`, но в рассматриваемом примере мы собираемся включить в представление другую таблицу, точнее, четыре таблицы.

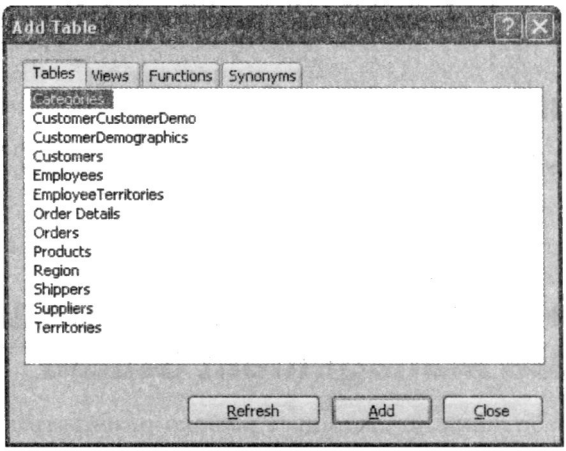


Рис. 10.2. Диалоговое окно Add Table

Чтобы выбрать одну таблицу, переведите на нее подсветку, а для выбора следующих таблиц щелкните на их названиях, удерживая нажатой клавишу <Ctrl>. Теперь щелкните на обозначении таблицы `Customers`, а затем нажмите и удерживайте нажатой клавишу <Ctrl>, выбирая таблицы `Orders`, `Order Details` и `Products`. В конечном итоге обозначения всех этих четырех таблиц должны быть выделены подсветкой, как показано на рис. 10.3.

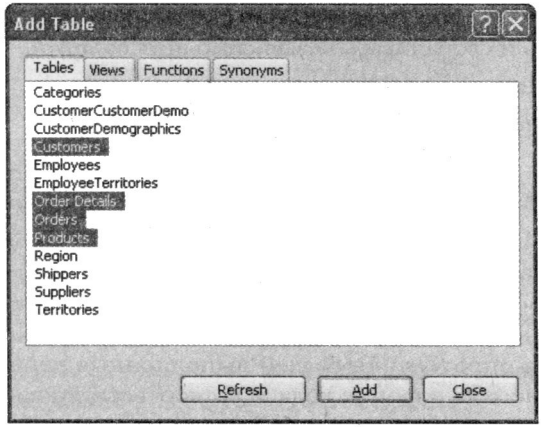


Рис. 10.3. Диалоговое окно Add Table после выбора нескольких таблиц

Затем щелкните на кнопке Add, чтобы СУБД SQL Server добавила несколько таблиц к представлению (в действительности в большинстве систем должна быть предусмотрена возможность увидеть, как происходит добавление этих таблиц в редакторе представлений, который будет рассмотрен ниже).

Прежде чем закрыть диалоговое окно Add Table, обратите внимание на вкладки Views, Functions и Synonyms, находящиеся в его верхней части. В представлении могут быть предусмотрены непосредственные ссылки на эти объекты, и данное диалоговое окно предоставляет возможность сразу же ввести эти ссылки в определение предоставления. А пока щелкните на кнопке Add и понаблюдайте за тем, как открывается окно редактора представлений, показанное на рис. 10.4.

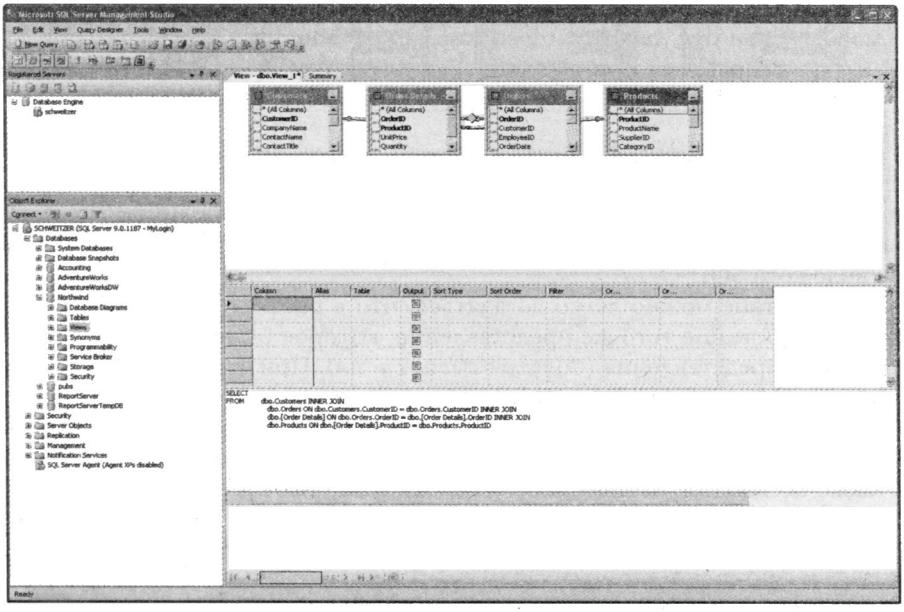


Рис. 10.4. Окно редактора представлений

Окно View Builder состоит из четырех перечисленных ниже подокон, которые можно разворачивать и сворачивать независимо друг от друга.

- Подокно Diagram.
- Подокно Criteria.
- Подокно SQL.
- Подокно Results.

Программисты, которым довелось работать с СУБД Access, могут заметить, что с помощью подокна Diagram выполняются примерно такие же действия, как и в окне редактирования запросов Access. В нем можно добавлять и удалять таблицы и даже определять связи. После каждого добавления таблиц, проставления отметок напротив имен столбцов и определения связей выполненные действия автоматически отражаются в подокне SQL в форме кода SQL, соответствующего разрабатываемой диаграмме. Чтобы определить назначение каждой из пиктограмм на панели инструментов, следует задержать над ней на короткое время указатель мыши, чтобы всплыла подсказка, описывающая назначение интересующей вас кнопки.

> Для того чтобы добавить таблицы, либо щелкните правой кнопкой мыши в подокне Diagram (см. рис. 10.4, *сверху*) и выберите команду Add Table во всплывающем меню, либо щелкните на кнопке Add table панели инструментов (эта кнопка обозначена пиктограммой со стрелкой вправо в левой верхней части).

После этого выберите некоторые столбцы, как показано на рис. 10.5.

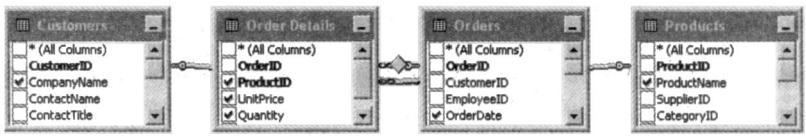


Рис. 10.5. Столбцы, выбранные в подокне Diagram

На рис. 10.5 ради экономии места показано только подокно Diagram. Если бы на экране во время выполнения указанных действий было развернуто подокно Grid, то в данном подокне появлялась бы информация о каждом выбранном столбце. Если же развернуто подокно SQL, то результаты формирования диаграммы отображаются также в виде кода SQL.

На данном этапе можно легко догадаться, что в рассматриваемом примере происходит формирование того же представления, которое было создано в первом примере сложного представления (`CustomerOrders_vw`). При этом единственной нетривиальной задачей является формирование вычисленного столбца (`ExtendedPrice`). Чтобы решить эту задачу, необходимо либо ввести вручную формулу вычисления значений этого столбца в подокне SQL, либо задать ее в столбце Column подокна Grid, указав также псевдоним вычисленного столбца (рис. 10.6).

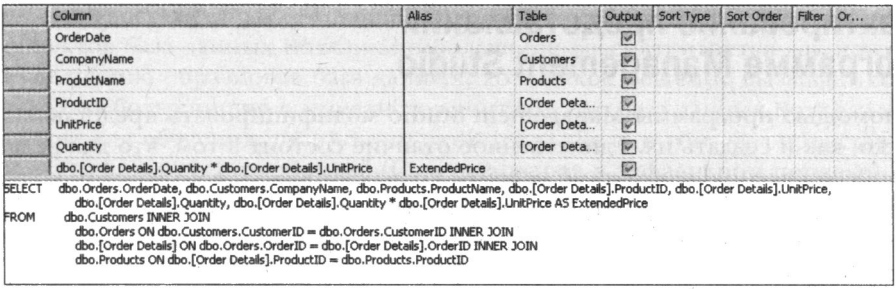


Рис. 10.6. Формула, введенная в столбце Column подокна Grid

После осуществления всех описанных действий в программе View Builder должен быть сформирован следующий код SQL:

```
SELECT dbo.Orders.OrderDate,
       dbo.Customers.CompanyName,
       dbo.Products.ProductName,
       dbo.[Order Details].ProductID,
       dbo.[Order Details].UnitPrice,
       dbo.[Order Details].Quantity,
       dbo.[Order Details].Quantity * dbo.[Order Details].UnitPrice AS
ExtendedPrice
FROM   dbo.Customers INNER JOIN
       dbo.Orders ON dbo.Customers.CustomerID = dbo.Orders.CustomerID INNER JOIN
       dbo.[Order Details] ON dbo.Orders.OrderID =
       dbo.[Order Details].OrderID INNER JOIN
       dbo.Products ON dbo.[Order Details].ProductID =
       dbo.Products.ProductID
```

Безусловно, этот код приведен в другом формате по сравнению с составленным вручную кодом сложного запроса, но его анализ показывает, что между кодом, сформированным автоматически и вручную, нет существенных различий!

> Если освоение синтаксиса запросов T-SQL дается вам с большим трудом, то вы можете использовать описанное инструментальное средство, чтобы проще понять синтаксис запросов. Достаточно перетащить определения нескольких таблиц в подокно Diagram, выбрать из каждой таблицы столбцы, которые должны быть включены в запрос, после чего в большинстве случаев программа Management Studio успешно сформирует необходимый запрос. Вам останется лишь изучить синтаксис кода, сформированного программой View Builder, чтобы узнать, как в следующий раз сформировать этот запрос без посторонней помощи.

Теперь перейдем к следующему этапу — сохраним полученное представление под именем `CustomerOrders2_vw` (автор предпочитает использовать для этой цели пиктограмму на панели инструментов с изображением диска) и закроем программу View Builder.

Редактирование представлений в программе Management Studio

С помощью программы Management Studio модифицировать представления так же легко, как и создать их. Единственное отличие состоит в том, что для редактирования представления необходимо перейти по дереву объектов к какому-то конкретному представлению, щелкнуть на его обозначении правой кнопкой мыши и выбрать во всплывающем меню команду Modify. После выполнения всех этих действий пользователя приветствует та же дружественная программа Query Designer, которая использовалась при создании запроса.

Просмотр и контроль существующего кода

На практике часто возникает необходимость ознакомиться с определением одного из представлений, которые применяются в базе данных. После изучения сведений, изложенных в предыдущих разделах, должно быть очевидно, что наиболее удобный способ решения этой задачи состоит в использовании программы Management Studio для выполнения тех же действий, которые осуществляются для редактирования представления. Перейдите в дереве объектов к подузлу Views, выберите обозначение представления, с которым вы хотите ознакомиться, щелкните на этом обозначении правой кнопкой мыши и выберите во всплывающем меню команду Modify View. Откроется окно, в котором будет показан код определения представления с выделенными разным цветом синтаксическими конструкциями.

Но, к сожалению, для ознакомления с кодом представления не всегда имеется возможность воспользоваться программой Management Studio (безусловно, кроме этой программы имеются другие инструментальные средства с меньшими возможностями, но они здесь не рассматриваются). Тем не менее для ознакомления с фактическим определением представления можно воспользоваться следующими вариантами:

- вызвать на выполнение процедуру `sp_helptext`;
- открыть системную таблицу `syscomments`.

Наиболее предпочтительный вариант состоит в использовании процедуры `sp_helptext`, поскольку с каждым новым выпуском формат системных таблиц немного изменяется, но все эти изменения учитываются в коде процедуры.

Рассмотрим пример применения процедуры `sp_helptext` для ознакомления с текстом одного из представлений, предусмотренных в базе данных `Northwind`, — представления "`Alphabetical list of products`":

```
EXEC sp_helptext [Alphabetical list of products]
```

СУБД SQL Server выводит на экран следующий код этого представления:

```
Text
----------------------------------------------------------------
create view "Alphabetical list of products" AS
SELECT Products.*, Categories.CategoryName
FROM Categories INNER JOIN Products ON Categories.CategoryID = Products.CategoryID
WHERE (((Products.Discontinued)=0))
```

Автор должен признаться, что испытал удовольствие, обнаружив в базе данных `Northwind` это представление, относящееся к числу наиболее необычных примеров,

когда-либо предоставленных компанией Microsoft (вероятно, именно из-за наличия таких объектов база данных Northwind больше не рассматривается как применяемая по умолчанию образцовая база данных, хотя такое решение вызывает сожаление, поскольку большинство специалистов считают, что база данных Northwind лучше подходит для обучения, чем AdventureWorks). Дело в том, что представление "Alphabetical list of products" (Список товаров в алфавитном порядке) не совсем соответствует своему названию. По мнению автора, причина этого заключается в том, что база данных Northwind перенесена из среды СУБД Access, в которой действительно выполнялась сортировка данных, полученных с помощью представления. А в языке SQL не предусмотрена возможность применения в представлениях операции, которая с виду кажется очень простой, — сортировки с помощью конструкции ORDER BY.

Исключением из этого правила, касающегося использования конструкции ORDER BY, является то, что ORDER BY может применяться при условии использования при этом также предиката TOP. Корпорация Microsoft оставила в базе данных Northwind определение рассматриваемого представления, имя которого говорит о том, что его результаты представлены в алфавитном порядке, но фактически получение таких результатов не гарантируется. В действительности после вызова на выполнение запроса к этому представлению есть шансы получить его результаты не в алфавитном порядке!

> Следует отметить, что запрет на использование конструкции ORDER BY распространяется только на код представления. После создания представления конструкция ORDER BY может использоваться в самом запросе, который обращается к этому представлению.

Теперь рассмотрим еще один способ ознакомления с кодом определения представления, основанный на использовании таблицы syscomments. Но следует учитывать, что при использовании таблицы syscomments (а также, применительно к рассматриваемой теме, большинства других системных таблиц) приходится не только учитывать проблемы совместимости, но и сталкиваться с дополнительными трудностями, обусловленными тем, что для обозначения всех объектов в системных таблицах используются идентификаторы объектов.

> В самой СУБД SQL Server для обозначения объектов применяются идентификаторы объектов. Это — целочисленные значения, и в этом состоит отличие идентификаторов объектов от имен, которые применяют пользователи для обозначения создаваемых ими объектов. Вообще говоря, описание идентификаторов объектов выходит за рамки данной книги, но читателю нужно знать об их существовании, поскольку такие идентификаторы можно встретить в сценариях, используемых другими разработчиками, или столкнуться с ними, глубоко изучая среду SQL.

К счастью, в данном случае указанную проблему можно обойти, выполнив соединение с таблицей sysobjects:

```
SELECT sc.text
FROM syscomments sc
JOIN sysobjects so
  ON sc.id = so.id
WHERE so.name = 'Alphabetical list of products'
```

И в этом случае формируется тот же блок кода (действительно, в данном случае действия, выполняемые в процедуре sp_helptext, сводятся к применению аналогичного запроса):

```
Text
------------------------------------------------------------------
create view "Alphabetical list of products" AS
SELECT Products.*, Categories.CategoryName
FROM Categories INNER JOIN Products ON Categories.CategoryID = Products.CategoryID
WHERE (((Products.Discontinued)=0))
(1 row(s) affected)
```

Автор еще раз настоятельно рекомендует избегать использования системных таблиц, но вместе с тем стремится раскрыть в данной книге все возможности.

Защита кода представлений с помощью шифрования

При создании какого-либо коммерческого программного продукта часто возникает необходимость защитить исходный код от постороннего взгляда. Прежде всего, такая необходимость касается представлений.

Чтобы зашифровать код представления, необходимо включить в оператор создания представления ключевое слово WITH ENCRYPTION. Но если в определении представления используется конструкция WITH CHECK OPTION, то приходится учитывать несколько дополнительных требований, описанных ниже.

- Ключевое слово WITH ENCRYPTION должно быть введено после имени представления, но перед ключевым словом AS.
- С ключевым словом WITH ENCRYPTION не должно использоваться ключевое слово OPTION.

Кроме того, следует помнить, что в случае модификации представления с помощью оператора ALTER VIEW происходит полная замена в базе данных существующего представления и остаются неизменными только назначенные права доступа. Это означает, что при этом из базы данных удаляется также зашифрованный текст представления. Поэтому если требуется, чтобы модифицированное представление также было зашифровано, то конструкция WITH ENCRYPTION должна быть предусмотрена и в операторе ALTER VIEW.

Рассмотрим пример применения оператора ALTER VIEW для модификации представления CustomerOrders_vw, которое было создано нами ранее в базе данных Northwind. Если же читатель не выполнил упражнение по созданию представления CustomerOrders_vw, то может просто заменить в следующем коде ключевое слово ALTER словом CREATE (напоминаем, что следующий оператор должен быть выполнен применительно к базе данных Northwind):

```
ALTER VIEW CustomerOrders_vw
WITH ENCRYPTION
AS
SELECT   cu.CompanyName,
         o.OrderDate,
         od.ProductID,
```

```
        p.ProductName,
        od.Quantity,
        od.UnitPrice,
        od.Quantity * od.UnitPrice AS ExtendedPrice
FROM    Customers AS cu
INNER JOIN  Orders AS o
    ON cu.CustomerID = o.CustomerID
INNER JOIN  [Order Details] AS od
    ON o.OrderID = od.OrderID
INNER JOIN  Products AS p
    ON od.ProductID = p.ProductID
```

После выполнения этого оператора предпримем попытку ознакомиться с определением представления CustomerOrders_vw с помощью процедуры sp_helptext:

```
EXEC sp_helptext CustomerOrders_vw
```

СУБД SQL Server сразу же сообщает, что не может выполнить это требование:

```
The object comments have been encrypted.
```

Допустим, что, вспомнив о возможности воспользоваться таблицей syscomments, мы попытаемся ознакомиться с текстом представления с ее помощью:

```
SELECT sc.text FROM syscomments sc
JOIN sysobjects so
  ON sc.id = so.id
WHERE so.name = 'CustomerOrders_vw'
```

Но и эта попытка не принесет желаемых результатов, поскольку СУБД SQL Server распознает, что таблица зашифрована, и выдаст результат NULL.

Короче говоря, зашифрованный код представления становится недоступным. Безуспешными являются даже попытки получить к нему доступ с помощью других средств просмотра (таких как программа Management Studio, которая фактически даже не обозначает как доступную команду Modify, относящуюся к зашифрованному представлению).

> Прежде чем привести в действие опцию WITH ENCRYPTION, обязательно сохраните в надежном месте исходный код представления, поскольку возможность извлечь зашифрованный код из базы данных исключена. Если окажется, что резервная копия кода, хранящаяся где-либо в другом месте, отсутствует, а в определение представления необходимо внести изменение, то вам придется написать это определение полностью заново.

Связывание представления со схемой

Процедура связывания представления со схемой позволяет непосредственно определить, от каких объектов (таблиц или других представлений) зависит рассматриваемое представление, а затем связать их с этим представлением. Важность данной процедуры заключается в том, что она позволяет воспрепятствовать внесению изменений (с помощью операторов CREATE и ALTER) в объекты, от которых зависит представление. Единственным способом уничтожения такой связи является удаление представления, связанного со схемой.

Причины, по которым может потребоваться применить представление. связанное со схемой, перечислены ниже.

- Связывание представления со схемой исключает возможность того, что представление станет "зависшим" в результате изменений в основополагающих объектах. Например, может оказаться, что в базе данных будет удален какой-то объект в результате выполнения оператора `DROP` или будут внесены какие-то другие изменения (вплоть до удаления столбца. от которого зависит работа представления), но при этом не будут учтены последствия, связанные с применением представления. Если же представление обозначено как `Schema Bound` (Связанное со схемой), этого не может произойти.
- Для обеспечения создания индексированного представления. Если на представлении должен быть задан индекс, то представление необходимо создать с помощью опции `SCHEMABINDING`. (Индексированные представления рассматриваются ниже.)
- Если должна быть создана пользовательская функция, связанная со схемой (а некоторые пользовательские функции обязательно должны быть связанными со схемой), которая ссылается на представление, то и это представление должно быть связанным со схемой.

Таким образом, при создании представления необходимо также рассмотреть вопрос о том, должно ли это представление быть связанным со схемой.

Придание представлению признаков таблицы с помощью опции VIEW_METADATA

Применение опции `VIEW_METADATA` приводит к тому, что в клиентских программах DB-LIB, ODBC и OLE-DB представление становится похожим на настоящую таблицу. Если эта опция не используется, то в клиентский API-интерфейс передаются метаданные, которые указывают на то, на какой базовой таблице (таблицах) основано это представление.

Предоставление информации о метаданных требуется для обеспечения возможности сделать обновляемыми все клиентские курсоры (курсоры, которыми управляют клиентские приложения). Следует отметить, что если предусмотрена поддержка таких курсоров, то следует также предусмотреть применение триггера `INSTEAD OF`.

Индексированные (материализованные) представления

В версии SQL Server 2000 возможность использования индексированных представлений предусматривалась только в варианте Enterprise Edition (несомненно, они поддерживались также в Developer Edition и Evaluation Edition, но применение этих вариантов программного обеспечения SQL Server, предназначенных для разработки и опробования кода, не допускалось в системах производственного назначения). Но после выпуска программы SQL Server 2005 индексированные представления поддерживаются во всех версиях.

Если представление используется в запросе, то код определения представления встраивается в код вызывающего запроса. К сожалению, это означает, что происходит значительное усложнение кода вызывающего запроса. Фактически дополнительные издержки, связанные с тем, что в СУБД приходится динамически анализировать действия, предусмотренные в представлении (и определять, на какие данные распространяются эти действия), могут становиться весьма значительными. Более того, часто обнаруживается, что операции соединения таблиц, предусмотренные в определении представления, вызывают необходимость выполнения дополнительных операций соединения таблиц в запросе. Индексированные представления позволяют устранять причины этих недостатков заранее, еще до вызова запроса на выполнение.

Индексированное представление – это такое представление, к которому относится набор уникальных значений, "материализованный" в форме кластеризованного индекса. Преимущество использования индексированных представлений состоит в том, что они обеспечивают очень быстрый поиск благодаря тому, что информация, лежащая в основе представления, уже собрана заранее. После создания на представлении первого индекса (который должен представлять собой кластеризованный индекс, сформированный на уникальном наборе значений) для СУБД SQL Server появляется также возможность создавать на этом представлении дополнительные индексы; при этом в качестве справочной информации используется кластеризованный ключ из первого индекса. Несмотря на то, что эти возможности весьма привлекательны, за них также приходится платить, поскольку при рассмотрении возможности создания индексов на представлениях необходимо учитывать некоторые требования, перечисленные ниже (список этих требований довольно велик).

- Представление должно быть создано с использованием опции `SCHEMABINDING`.
- Если в представлении имеется ссылка на какие-либо пользовательские функции (дополнительная информация на эту тему приведена ниже), то эти функции также должны быть связанными со схемой.
- Представление не должно ссылаться на какие-либо другие представления; допускается использование только ссылок на таблицы и пользовательские функции.
- Имена всех таблиц и пользовательских функций, на которые имеется ссылка в представлении, должны быть основаны на применении соглашения об именовании, предусматривающего двухкомпонентную структуру имен, например `dbo.Customers`, `BillyBob.SomeUDF` (не допускаются даже обычные трех- и четырехкомпонентные имена); кроме того, эти объекты должны иметь того же владельца, что и представление.
- Определение представления должно находиться в той же базе данных, что и все объекты, на которые имеется ссылка в представлении.
- Ко времени создания всех основополагающих таблиц самого представления опциям `ANSI_NULLS` и `QUOTED_IDENTIFIER` должно быть присвоено значение `on` (с помощью команды `SET`).
- Все функции, на которые ссылается представление, должны быть детерминированными.

Прежде чем перейти к рассмотрению примера индексированного представления, внесем некоторые изменения в объект CustomerOrders_vw, который был создан ранее в этой главе:

```
ALTER VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT   cu.CompanyName,
         o.OrderID,
         o.OrderDate,
         od.ProductID,
         p.ProductName,
         od.Quantity,
         od.UnitPrice
FROM     dbo.Customers AS cu
INNER JOIN   dbo.Orders AS o
     ON cu.CustomerID = o.CustomerID
INNER JOIN   dbo.[Order Details] AS od
     ON o.OrderID = od.OrderID
INNER JOIN   dbo.Products AS p
     ON od.ProductID = p.ProductID
```

Ниже перечислены основные изменения, на которые следует обратить внимание в этом коде.

- Представление создается с опцией SCHEMABINDING.
- Чтобы иметь возможность использовать опцию SCHEMABINDING, был осуществлен переход к применению двухкомпонентной схемы именования всех объектов, на которые ссылается представление (в данном случае всех таблиц).

Из определения представления пришлось удалить вычисленный столбец; безусловно, создание индексированных представлений с выражениями, не требующими агрегирования данных, допускается, но оптимизатор запросов их игнорирует.

Но фактически выполнение этого оператора представляет собой лишь первый шаг, поскольку он еще не приводит к созданию индексированного представления. Вместо этого было получено представление, которое может быть индексировано. А когда мы приступим к созданию индексов, необходимо учитывать, что первый индекс, созданный на представлении, должен быть одновременно и кластеризованным, и уникальным:

```
CREATE UNIQUE CLUSTERED INDEX ivCustomerOrders
ON CustomerOrders_vw(CompanyName, OrderID, ProductID)
```

После вызова на выполнение этого оператора представление становится кластеризованным. Но к этому моменту появляется небольшая проблема, которая вскоре станет очевидной.

Проверим полученное представление, применив к нему простой оператор SELECT:

```
SELECT * FROM CustomerOrders_vw
```

После вызова этого оператора на выполнение из командной строки создается впечатление, что все в порядке, но при попытке сформировать графический план запроса, как показано на рис. 10.7, обнаруживается проблема (пиктограмма получения гра-

фического плана запроса обозначается всплывающей подсказкой `Display Estimated Execution Plan` и находится примерно в середине панели инструментов).

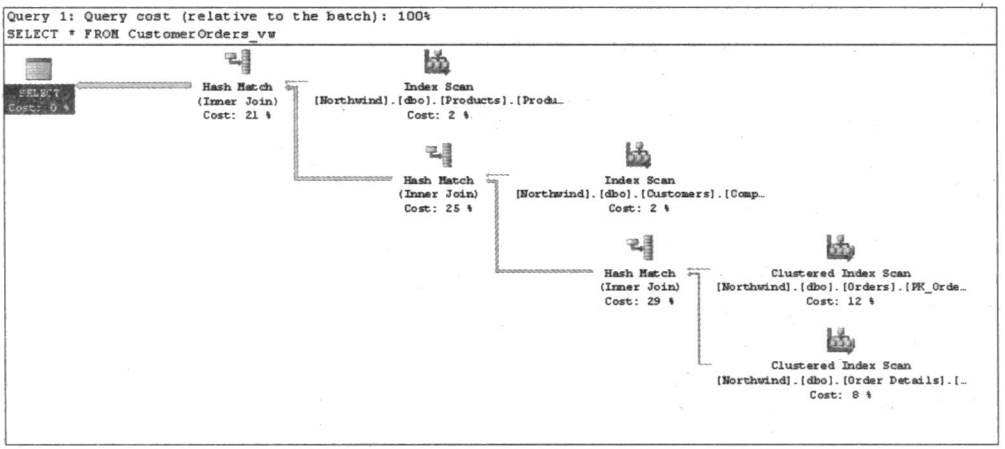


Рис. 10.7. Графический план запроса

Признаки небольшой проблемы, о которой шла речь выше, обнаруживается в графическом плане запроса, показанном на рис. 10.7. Дело в том, что ни одна часть этого плана не показывает, что в каком-либо виде используется созданный нами индекс!

Но фактически эта проблема обусловлена лишь небольшими размерами применяемых нами таблиц. Дело в том, что в базе данных `Northwind` нет достаточного количества данных. Этот пример показывает, что оптимизатор находит компромисс, определяя, сколько времени потребуется на выполнение первого найденного им плана, и какой объем работы должен быть им выполнен для поиска лучшего плана. Например, имеет ли смысл затрачивать еще две секунды на составление лучшего плана, если план, который уже известен, может быть выполнен за одну секунду!

В рассматриваемом примере оптимизатор СУБД SQL Server рассматривает основополагающую таблицу, обнаруживает, что в ней не так уж много данных, и решает, что полученный план уже "достаточно хорош", поэтому нет смысла продолжать исследование для определения того, не будет ли более быстродействующим план, в котором используется индекс, созданный на представлении.

> *Отметим, что оптимизатор, принимая решение о том, следует ли продолжать поиск лучшего плана, учитывает количество строк в таблице, рассматривая любой индекс, а не только индекс, заданный на представлении. Если набор данных невелик, то вероятность полного игнорирования индекса оптимизатором SQL Server в пользу первого же обнаруженного плана становится весьма значительной. В подобных случаях приходится нести издержки, связанные с сопровождением индекса (что выражается в замедлении выполнения операторов `INSERT`, `UPDATE` и `DELETE`), не получая никаких преимуществ в форме ускоренного выполнения операторов `SELECT`.*

Тем не менее, чтобы ознакомиться с возможностями, которые предоставляют индексированные представления, создадим базу данных, имеющую достаточный объем данных для того, чтобы индекс стал более привлекательным для оптимизатора. Читатель может загрузить с сопровождающего узла книги и выполнить сценарий заполнения базы данных, называемый `CreateAndLoadNorthwindBulk.sql`.

Следует учитывать, что если по умолчанию выполняется загрузка того количества данных, которое предусмотрено в этом сценарии, то для базы данных `NorthwindBulk` *потребуется около 55 Мбайт дискового пространства. Более того, следует иметь в виду, что на выполнение этого сценария заполнения базы данных может потребоваться значительное время, так как с его помощью формируются и загружаются тысячи и тысячи строк данных.*

После этого достаточно просто снова создать представление и индекс в новой базе данных `NorthwindBulk`, как показано ниже.

```
USE NorthwindBulk
GO

CREATE VIEW CustomerOrders_vw
WITH SCHEMABINDING
AS
SELECT   cu.CompanyName,
         o.OrderID,
         o.OrderDate,
         od.ProductID,
         p.ProductName,
         od.Quantity,
         od.UnitPrice
FROM     dbo.Customers AS cu
INNER JOIN   dbo.Orders AS o
     ON cu.CustomerID = o.CustomerID
INNER JOIN   dbo.[Order Details] AS od
     ON o.OrderID = od.OrderID
INNER JOIN   dbo.Products AS p
     ON od.ProductID = p.ProductID
GO

CREATE UNIQUE CLUSTERED INDEX ivCustomerOrders
ON CustomerOrders_vw(CompanyName, OrderID, ProductID)
```

На этот раз повторно вызовем на выполнение первоначальный запрос, но применительно к базе данных `NorthwindBulk`:

```
USE NorthwindBulk

SELECT * FROM CustomerOrders_vw
```

Проверим новый план запроса (рис. 10.8).

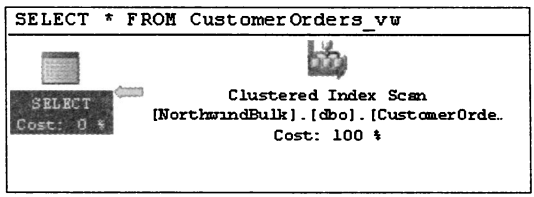


Рис. 10.8. План запроса, сформированный с учетом индекса

На этот раз оптимизатору СУБД SQL Server приходится иметь дело с объемом данных, достаточным для более тщательного составления плана запроса. В данном случае оптимизатор принимает во внимание наличие индексированного представления, которое определено на рассматриваемой таблице. Теперь общая производительность этого представления становится намного выше (в расчете на отдельную строку) по сравнению с предыдущим вариантом его использования.

Резюме

Практика показывает, что во многих базах данных представления применяются либо слишком широко, либо недостаточно полно. В частности, есть такие разработчики, которые, по-видимому, стремятся оформить в виде представлений весь код доступа к базе данных (не учитывая того, что в результате происходит переход на более высокий уровень абстракции и количество промежуточных этапов доступа к данным увеличивается). Встречаются и такие специалисты по базам данных, которые, очевидно, вообще забывают о существовании таких средств обработки данных, как представления. Сам автор считает, что представления, как и все прочие программные средства, должны применяться исключительно исходя из того, насколько они подходят для решения конкретной задачи, не в большей, не в меньшей степени. При использовании представлений необходимо учитывать приведенные ниже соображения.

- Не рекомендуется создавать представления на основе других представлений. Вместо этого во вновь создаваемое представление необходимо включить соответствующую информацию запроса из исходного представления.
- Следует помнить, что представления, в которых используется опция `WITH CHECK OPTION`, предоставляют определенные функциональные возможности, которые не могут быть реализованы с помощью обычного ограничения `CHECK`.
- Представления, исходный код которых не должен становиться доступным для просмотра посторонними, могут быть зашифрованы. Решение о шифровании представлений обычно принимается с учетом того, имеет ли создаваемый программный продукт коммерческое назначение, или исходя из общих соображений обеспечения защиты информации.
- Если для внесения изменений в представление используется оператор `ALTER VIEW`, то существующее представление полностью заменяется (хотя определенные ранее права доступа к представлению остаются неизменными). Из этого следует, что если требуется, чтобы в модифицированном представлении продолжали действовать заданные ранее опции кодирования и ограничения, то в оператор `ALTER VIEW` необходимо включить конструкции `WITH ENCRYPTION` и `WITH CHECK OPTION`.
- Для отображения кода, лежащего в основе представления, следует использовать системную процедуру `sp_helptext`; не рекомендуется для ознакомления с этим кодом обращаться к системным таблицам.
- Применение представлений для выполнения запросов производственного назначения необходимо свести к минимуму, поскольку они вносят дополнительные издержки и вызывают снижение производительности.

Основные области использования представлений перечислены ниже.

- Выборка строк с учетом заданных критериев.
- Защита конфиденциальных данных.
- Уменьшение кажущейся сложности базы данных.
- Создание дополнительного уровня доступа, в результате чего несколько физических баз данных объединяются в одну логическую базу данных.

В следующей главе приведены сведения об использовании пакетов и создании сценариев. Определенная информация по этой теме приведена и в данной главе, в связи с описанием сценария с операторами `INSERT`, предназначенного для вставки строк в таблицу `Orders`, в котором информация из вновь вставленной строки затем использовалась для вставки данных в таблицу `Order Details`. Пакеты и сценарии служат основой для создания хранимых процедур — программных объектов СУБД SQL Server, которые в наибольшей степени соответствуют определению собственных программ СУБД.

Упражнения

10.1. Создайте в базе данных `Northwind` представление `Managers`, с помощью которого можно получить данные только о тех служащих, которые выполняют обязанности руководителей по отношению к другим служащим.

10.2. Модифицируйте представление, созданное по условиям упражнения 10.1, так, чтобы оно стало зашифрованным.

10.3. Повторно создайте и индексируйте существующее представление "`Products by Category`" базы данных `Northwind` на основе столбцов `CategoryName` и `ProductName`.

Сценарии и пакеты

В предыдущих главах было приведено много **сценариев** SQL, а настоящая глава полностью посвящена проблематике создания сценариев. Сценарий может состоять из одного или нескольких операторов SQL, в том числе `CREATE`, `ALTER`, `SELECT` и т.д. Тем не менее сценарии, состоящие из одного оператора, встречаются редко, поскольку достаточно сложная работа может быть выполнена лишь с помощью сценария, который включает в свой состав большое количество операторов.

В сценариях SQL, состоящих из нескольких операторов, фактически осуществляется целый ряд взаимосвязанных действий, и это позволяет решать очень сложные задачи. А если в состав сценария, кроме операторов SQL, входят языковые элементы, поддерживаемые инфраструктурой .NET, то доступные возможности расширяются еще больше.

Обычно сценарии предназначены для решения какой-то определенной задачи. Иными словами, каждый оператор, входящий в состав сценария, направлен на достижение некоторой общей цели. В качестве примеров можно указать сценарии, предназначенные для создания базы данных (к такому типу относятся сценарии, предназначенные для инсталляции системы), сценарии, используемые в процессе сопровождения системы (утилиты резервного копирования, утилиты DBCC), и т.д. Таким образом, сценарий может состоять из любых операторов, которые обычно выполняются в связи друг с другом.

В настоящей главе рассматриваются не только сценарии, но и **пакеты**. Пакет представляет собой средство управления способом группирования операторов в СУБД SQL Server. Кроме того, в данной главе будет описана утилита **SQLCMD** с интерфейсом командной строки и показано, какое отношение она имеет к сценариям.

Утилита SQLCMD была впервые введена в версии SQL Server 2005. В этой версии SQL Server поддерживается также программа `osql.exe`, которая применялась в предыдущих версиях для работы с интерфейсом командной строки (но такая поддержка предусмотрена лишь для обеспечения обратной совместимости). Кроме того, в документации можно встретить

упоминание о программе `isql.exe` *(которую не следует путать с* `isqlw.exe`*); программа* `isqlw.exe` *выполняла в предыдущих версиях такое же назначение, как и* `osql.exe`*. Но начиная с версии SQL Server 2005, программа* `isql.exe` *больше не поддерживается.*

Основные сведения о сценариях

С формальной точки зрения сценарием считается совокупность операторов, хранящихся в виде отдельного файла, который может вызываться на выполнение и использоваться повторно. Сценарии SQL хранятся в виде текстовых файлов. В программе SQL Server Management Studio предусмотрен целый ряд инструментальных средств, позволяющих упростить написание сценариев. В основном окне ввода запросов этой программы применяется цветовое выделение, которое позволяет не только распознавать ключевые слова, но и определять их назначение. Кроме того, в программе Management Studio предусмотрен пошаговый отладчик, даны образцы кода, имеется браузер объектов и т.д.

Сценарии обычно рассматриваются как отдельная единица работы. Это означает, что в обычных обстоятельствах сценарий либо выполняется полностью, либо вообще не выполняется. В сценариях могут использоваться и системные, и локальные переменные. В качестве примера рассмотрим сценарий, который применялся для вставки строк о заказах с помощью оператора `INSERT` в главе, посвященной представлениям (глава 10):

```
USE Northwind
DECLARE @Ident int
INSERT INTO Orders
(CustomerID,OrderDate)
VALUES
('ALFKI', DATEADD(day,-1,GETDATE()))
SELECT @Ident = @@IDENTITY
INSERT INTO [Order Details]
(OrderID, ProductID, UnitPrice, Quantity)
VALUES
(@Ident, 1, 50, 25)

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident)
```

В этом сценарии применяются шесть различных операторов, которые показывают, какие разнообразные действия могут осуществляться в сценарии. В сценарии используются системные и локальные переменные, операторы `USE` и `INSERT`, а также две версии оператора `SELECT` — с операцией присваивания и обычная. Все эти языковые элементы взаимодействуют в целях осуществления общей задачи — вставки в базу данных заполненных заказов.

Оператор USE

Оператор `USE` сообщает, какая база данных должна стать текущей. Действие, осуществляемое этим оператором, отражается во всех местах сценария, в которых используются заданные по умолчанию значения для той части полностью уточненного имени объекта, которая применяется для обозначения имени базы данных. В данном конкретном примере не приводилась информация о том, в какой базе данных находятся таблицы, применяемые в операторе `INSERT` или `SELECT`, но поскольку в этот

сценарий оператор USE включен перед операторами INSERT и SELECT, в данных операторах используется указанная база данных (в рассматриваемом случае Northwind). А если бы оператор USE не был предусмотрен, то оставалось бы лишь надеяться на то, что пользователь, вызывающий сценарий на выполнение, позаботится об указании в качестве текущей именно той базы данных, которая требуется.

> Не следует рассматривать это пояснение как намек на то, что в сценарии всегда следует включать операторы USE, поскольку необходимость в использовании такого оператора зависит от назначения сценария. В частности, если сценарий предназначен для универсального применения, то фактически может потребоваться исключить оператор USE.
>
> Как правило, оператор USE требуется, если в сценарии используются таблицы какой-то конкретной базы данных (иными словами, таблицы, отличные от системных). По мнению автора, оператор USE становится также необходимым, если сценарий предназначен для модификации конкретной базы данных, — как было указано в предыдущих главах, не поддается описанию количество случаев, в которых непреднамеренно создавалось большое количество таблиц в базе данных master, тогда как эти таблицы были предназначены для пользовательской базы данных.

За оператором USE в рассматриваемом сценарии следует оператор DECLARE с объявлением переменной. Оператор DECLARE кратко упоминался в предыдущих главах, а в этой главе приведено более подробное его описание.

Объявление переменных

Оператор DECLARE имеет следующий довольно простой синтаксис:

```
DECLARE  @<variable name> <variable type>[,
       @<variable name> <variable type>[,
       @<variable name> <variable type>]]
```

С помощью оператора DECLARE можно одновременно объявить одну или несколько переменных. На практике часто приходится сталкиваться с тем, как в сценарии используется отдельный оператор DECLARE для каждой объявляемой переменной, вместо указания одного, разделенного запятыми списка переменных в одном операторе DECLARE. Безусловно, разработчик вправе сам выбирать тот или другой способ объявления переменных, но независимо от выбранного способа значение объявленной переменной всегда остается равным NULL, до тех пор, пока вместо него не будет задано какое-то другое значение.

В данном случае объявлена как целочисленная одна локальная переменная @ident. С формальной точки зрения в данном сценарии можно было бы обойтись без объявления этой переменной, поскольку вместо нее можно применять непосредственно значение @@IDENTITY. Отметим, что значение @@IDENTITY присваивается с помощью системной переменной. Это значение всегда доступно и представляет собой последнее идентификационное значение, присвоенное в текущем соединении. Но, как и при использовании большинства других системных переменных, следует усвоить привычку явно переносить значение @@IDENTITY в локальную переменную. Это позволяет гарантировать, что в само значение системной переменной не будут случайно внесены изменения. Безусловно, в рассматриваемом сценарии такая опасность вообще исключена, но, как всегда, рекомендуем соблюдать во всем единообразие.

Автор всегда предпочитает переносить значение, полученное с помощью любой системной переменной, в специально объявленную для этого переменную. Это позволяет мне с уверенностью использовать значение переменной, зная о том, что за любое изменение этого значения отвечаю я сам. Если же применяется непосредственно системная переменная, то зачастую нельзя быть уверенным в том, не произошло ли изменение значения, устанавливаемого с ее помощью, поскольку вызовы большинства системных переменных осуществляются не вами, а самой системой. В результате этого вполне могут возникнуть такие ситуации, что система изменяет значение, возвращаемое системной переменной, в тот момент, когда вы этого совсем не ожидаете, что приводит к появлению самых неприятных последствий, связанных с эксплуатацией приложения, — непредсказуемых результатов.

Задание значений переменных

В предыдущем разделе было показано, как объявить переменные, а в этом разделе речь пойдет о том, как изменить значение переменной. В настоящее время в языке SQL предусмотрены два способа задания значения переменной – для этой цели можно использовать оператор `SELECT` или `SET`. С точки зрения выполняемых функций эти операторы действуют почти одинаково, не считая того, что оператор `SELECT` позволяет получить исходное присваиваемое значение из таблицы, указанной в операторе `SELECT`.

Автор до сих пор не может понять, по каким причинам предусмотрены оба этих способа. Опубликовав две книги, в которых был задан этот вопрос, я надеялся, что кто-то пришлет мне письмо по электронной почте и даст хороший ответ, но этого не произошло. Достаточно сказать, что оператор `SET` теперь предусмотрен в стандарте ANSI, поэтому его описание приведено и в настоящей книге. Но я не мог найти никаких возражений против использования аналогичных функциональных средств, предусмотренных в операторе `SELECT`; создается впечатление, что даже в стандарте ANSI не исключается возможность присваивать литеральные значения с помощью оператора `SELECT`, а не `SET`. Я уверен, что оператор `SET` предусмотрен ради страховки, но не могу утверждать это категорически. На этом закончим анализ причин, по которым предусмотрены два оператора присваивания, и перейдем к рассмотрению различий в способах их использования.

Задание значений переменных с помощью оператора SET

Оператор `SET` обычно используется для задания значений переменных в такой форме, какая более часто встречается в процедурных языках. В качестве типичных примеров применения этого оператора можно указать следующие:

```
SET @TotalCost = 10
SET @TotalCost = @UnitCost * 1.1
```

Обратите внимание на то, что во всех этих операторах непосредственно осуществляются операции присваивания, в которых используются либо явно заданные значения, либо другие переменные. С помощью оператора `SET` невозможно присвоить переменной значение, полученное с помощью запроса; запрос должен быть выполнен отдельно и только после этого полученный результат может быть присвоен с помощью оператора `SET`. Например, попытка выполнения такого оператора вызывает ошибку:

```
USE Northwind
DECLARE @Test money
SET @Test = MAX(UnitPrice) FROM [Order Details]
SELECT @Test
```

а следующий оператор выполняется вполне успешно:

```
USE Northwind
DECLARE @Test money
SET @Test = (SELECT MAX(UnitPrice) FROM [Order Details])
SELECT @Test
```

Безусловно, последняя синтаксическая конструкция является вполне допустимой, но в соответствии с общепринятым соглашением такой способ реализации кода никогда не применяется. И в этом случае автор не может сказать со всей уверенностью, почему "так никто не делает", но подозревает, что в данном случае причиной является желание обеспечить удобство чтения. Дело в том, что обычно принято рассматривать операторы SELECT как предназначенные для выборки данных из таблиц, а операторы SET – как средства простого присваивания значений переменным.

Задание значений переменных с помощью оператора SELECT

Оператор SELECT обычно используется для присваивания значений переменным, если источником информации, которая должна быть сохранена в переменной, является запрос. Например, действия, осуществляемые в приведенном выше коде, гораздо чаще реализуются с помощью оператора SELECT:

```
USE Northwind
DECLARE @Test money

SELECT @Test = MAX(UnitPrice) FROM [Order Details]
SELECT @Test
```

Обратите внимание на то, что данный код немного понятнее (в частности, он более лаконичен, хотя и выполняет те же действия).

Итак, кратко сформулируем общепринятое соглашение по использованию того и другого оператора.

- Оператор SET используется, если должна быть выполнена простая операция присваивания значения переменной, т.е. если присваиваемое значение уже задано явно в форме определенного значения или в виде какой-то другой переменной.
- Оператор SELECT применяется, если присваивание значения переменной должно быть основано на запросе.

Автор готов принять упреки в том, что сам неоднократно нарушал это соглашение во многих примерах, приведенных в данной книге. Возможность использовать оператор SET для присваивания значений переменным впервые появилась в версии 7.0, и я вынужден признать, что еще не смог полностью принять на вооружение этот способ присваивания, даже несмотря на то, что после появления указанной версии прошло больше шести лет. Тем не менее создается впечатление, что компания Microsoft и сообщество пользователей SQL Server стремятся распространить действие данного соглашения, поэтому настоятельно рекомендуют начинающему разработчику стартовать с правильной позиции и неуклонно придерживаться этого соглашения.

Обзор системных переменных

В программном обеспечении СУБД SQL Server предусмотрено свыше тридцати системных переменных, не имеющих параметров. В табл. 11.1 приведено описание наиболее часто применяемых системных переменных.

Таблица 11.1. Наиболее часто применяемые системные переменные

Системная переменная	Назначение	Примечание
`@@CURSOR_ROWS`	Предоставляет информацию о том, сколько строк находится в текущее время в последнем результирующем наборе курсора, открытого в текущем соединении	Начиная с версии SQL Server 7 предусмотрена возможность заполнять курсоры асинхронно. Следует учитывать, что значение этой системной переменной может измениться, если курсор все еще находится в процессе заполнения
`@@DATEFIRST`	Предоставляет информацию о том, какой день в настоящее время считается первым днем недели (воскресенье, `Sunday`, или понедельник, `Monday`)	Задается в масштабах всей системы; поэтому, если кто-то изменит соответствующий системный параметр настройки для одного приложения, это отразится на работе всех приложений
`@@ERROR`	Содержит номер ошибки, возникшей при выполнении последнего оператора T-SQL в текущем соединении. Если ошибка не обнаружена, содержит `0`	Значение этой системной переменной переустанавливается после выполнения каждого очередного оператора. Если требуется сохранить содержащееся в ней значение, то это значение следует переносить в локальную переменную сразу же после выполнения оператора, для которого должен быть сохранен код ошибки
`@@FETCH_STATUS`	Эта системная переменная используется с оператором `FETCH`	Принимает значение `0`, если операция выборки строки оказалась допустимой, значение `-1`, если произошел выход за пределы результирующего набора курсора, и `-2`, если требуемая строка отсутствует (допустим, в связи с тем, что она удалена). Типичный вариант неправильной организации работы может быть обусловлен тем, что предполагается, будто любое ненулевое значение свидетельствует о выходе за пределы результирующего набора курсора, тогда как фактически значение `-2` может указывать на отсутствие отдельной строки
`@@IDENTITY`	Содержит последнее идентификационное значение, вставленное в базу данных в результате выполнения последнего оператора `INSERT` или `SELECT INTO`	Если в последнем операторе `INSERT` или `SELECT INTO` не произошла выработка идентификационного значения, системная переменная `@@IDENTITY` содержит `NULL`. Это утверждение остается справедливым, даже если отсутствие идентификационного значения было вызвано аварийным завершением при выполнении оператора. А если с помощью одного оператора осуществляется несколько операций вставки, этой системной переменной присваивается только последнее идентификационное значение

Окончание табл. 11.1

Системная переменная	Назначение	Примечание
`@@OPTIONS`	Предоставляет информацию об опциях SQL Server, которые были установлены с использованием команды `SET`	Значение системной переменной `@@OPTIONS` представляет собой двоичное число, биты которого обозначают отдельные опции SQL Server. Для проверки того, установлена ли конкретная опция, из значения `@@OPTIONS` (применив к нему соответствующий битовый флажок) необходимо выделить значение одного бита с помощью оператора побитовой обработки
`@@REMSERVER`	Системная переменная `@@REMSERVER` используется только в хранимых процедурах. Возвращает данные о сервере, с помощью которого вызвана хранимая процедура	Системная переменная `@@REMSERVER` применяется, если необходимо, чтобы в хранимой процедуре выполнялись различные действия, в зависимости от того, с какого удаленного сервера (часто находящегося в другом географическом местоположении по сравнению с клиентским приложением) была вызвана хранимая процедура. Тем не менее, если есть возможность для получения информации об удаленном сервере воспользоваться функциональными средствами .NET, так и следует делать, а не прибегать к использованию системной переменной `@@REMSERVER`
`@@ROWCOUNT`	Одна из наиболее широко используемых системных переменных. Возвращает информацию о количестве строк, затронутых последним оператором	Обычно применяется для контроля ошибок, отличных от тех, которые относятся к категории ошибок этапа прогона программы. Например, если в программе обнаруживается, что после вызова на выполнение оператора `DELETE` с конструкцией `WHERE` количество затронутых строк равно нулю, то можно сделать вывод, что произошло нечто непредвиденное. После этого сообщение об ошибке может быть активизировано вручную
`@@SERVERNAME`	Системная переменная `@@SERVERNAME` предоставляет информацию об имени локального сервера, с которого был запущен сценарий	Значение этой системной переменной можно изменить путем вызова на выполнение хранимой процедуры `sp_addserver` и последующего перезапуска СУБД SQL Server, но такая необходимость возникает редко
`@@TRANCOUNT`	Системная переменная `@@TRANCOUNT` предоставляет информацию о количестве активизированных транзакций для текущего соединения (которое по существу характеризует уровень вложенности транзакций)	Если не используются точки сохранения, то в результате выполнения оператора `ROLLBACK TRAN` значение `@@TRANCOUNT` уменьшается до нуля. В результате выполнения оператора `BEGIN TRAN` значение `@@TRANCOUNT` увеличивается на `1`, а в результате выполнения оператора `COMMIT TRAN` значение `@@TRANCOUNT` уменьшается на `1`
`@@VERSION`	Системная переменная `@@VERSION` предоставляет информацию о текущей версии SQL Server, а также о дате, процессоре и названии операционной системы	К сожалению, информация, содержащаяся в системной переменной `@@VERSION`, не представлена в каком-либо структурированном виде, с разбивкой на поля, поэтому, чтобы иметь возможность использовать какую-либо часть этой информации, необходимо предусмотреть синтаксический анализ строкового значения этой системной переменной

Не стоит беспокоиться, если описание некоторых системных переменных, приведенное в табл. 11.1, остается пока непонятным. В этой и в следующих главах даны все необходимые пояснения, поэтому в дальнейшем вы сможете использовать эту таблицу в основном для получения справочной информации. А на данный момент достаточно лишь отметить, что системные переменные предоставляют огромный объем информации о текущем состоянии системы и выполняемых в ней действиях.

Использование системной переменной @@IDENTITY

Системная переменная @@IDENTITY относится к числу наиболее важных из всех системных переменных. Напомним, что речь об идентификационных значениях шла в главе 5. В этой главе было указано, что при вставке данных не задается значение для столбца идентификации, так как СУБД SQL Server вставляет в этот столбец формируемое числовое значение автоматически.

В том случае, который рассматривался в качестве примера, значение @@IDENTITY было получено непосредственно после выполнения вставки в таблицу Orders. Дело в том, что при выполнении операции вставки мы не задаем значение ключа для таблицы; это значение формируется автоматически при выполнении операции вставки. Но вслед за тем, как в таблицу Orders будет введена информация о заказе, необходимо также ввести строку с данными о компонентах заказа в таблицу Order Details, но для этого требуется определить значение первичного ключа той строки таблицы Orders, которая связана со вставляемой строкой (напомним, что на таблице Order Details задано ограничение внешнего ключа, которое ссылается на таблицу Orders). Это значение ключа вырабатывает СУБД SQL Server, а не предоставляет пользователь, поэтому должен быть предусмотрен способ выборки данного значения для использования во время вставки в зависимые таблицы в ходе дальнейшего выполнения сценария. Такое автоматически формируемое значение ключа можно получить с помощью системной переменной @@IDENTITY, поскольку она возвращает идентификационное значение, сформированное в результате выполнения предыдущего оператора.

В рассматриваемом примере можно было бы легко обойтись без промежуточной передачи значения @@IDENTITY в локальную переменную, поскольку достаточно было бы просто явно сослаться на это значение в следующем операторе INSERT. Но автор усвоил привычку всегда передавать полученное значение в локальную переменную, чтобы избежать ошибок, которые могут возникать в тех случаях, если действительно нужна копия данных. В качестве примера такой ситуации можно указать применение еще одного оператора INSERT, зависящего от идентификационного значения, которое получено после применения оператора INSERT для вставки данных в таблицу Orders. Если бы это значение не было передано в локальную переменную, то после выполнения следующего оператора INSERT оно было бы потеряно. Дело в том, что после этой вставки данных в таблицу Order Details идентификационное значение перезаписывается, а поскольку в таблице Order Details нет столбца идентификации, то значение, полученное с помощью системной переменной @@IDENTITY, становится равным NULL. Кроме того, передача значения @@IDENTITY в локальную переменную позволяет сохранить это значение на тот случай, что придется в дальнейшем вывести его на печать для справок.

Чтобы ознакомиться с примером применения значения @@IDENTITY, создадим ряд таблиц:

```
CREATE TABLE TestIdent
(
  IDCol   int   IDENTITY
  PRIMARY KEY
)
CREATE TABLE TestChild1
(
  IDcol   int
  PRIMARY KEY
  FOREIGN KEY
        REFERENCES TestIdent(IDCol)
)
CREATE TABLE TestChild2
(
  IDcol   int
  PRIMARY KEY
  FOREIGN KEY
        REFERENCES TestIdent(IDCol)
)
```

Одна из таблиц, рассматриваемых в данном примере, является родительской. В ней имеется столбец идентификации, применяемый в качестве столбца первичного ключа (по стечению обстоятельств этот столбец в родительской таблице является единственным). Кроме того, предусмотрены две дочерние таблицы. Каждая из дочерних таблиц участвует в связи, основанной на использовании идентификатора. В подобных случаях в дочерней таблице по крайней мере часть первичного ключа (в рассматриваемом примере – весь первичный ключ) формируется на основе внешнего ключа, связанного с другой таблицей (родительской). А в данном примере обе дочерние таблицы должны получать значение своего ключа из родительской таблицы. Таким образом, необходимо вначале вставить строку в родительскую таблицу, а затем осуществить выборку сформированного при этом идентификационного значения, чтобы можно было использовать это значение для вставки данных в дочерние таблицы.

Практическое занятие Применение системной переменной `@@IDENTITY`

Теперь, после определения необходимых таблиц, можно приступить к написанию требуемого проверочного сценария:

```
/*************************************
* Этот сценарий показывает, что значение  *
* идентификации после выдачи еще одного   *
* оператора INSERT теряется               *
*************************************/
DECLARE @Ident   int -- Эта переменная предназначена для хранения информации.
                     -- На ее примере показано, как перемещать значения,
                     -- полученные из системных функций, в безопасное место
INSERT INTO TestIdent
       DEFAULT VALUES
SET @Ident = @@IDENTITY
PRINT 'The value we got originally from @@IDENTITY was ' +
       CONVERT(varchar(2),@Ident)
PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY
)
```

```
/* Выполнение первого оператора INSERT с использованием значения @@IDENTITY
** оканчивается успешно. Это значение остается неизменным, поскольку между
** первоначальным оператором INSERT и текущим нет других операторов. Но
** выполнение следующего оператора INSERT становится не столь успешным */
INSERT INTO TestChild1
VALUES
     (@@IDENTITY)
PRINT 'The value we got originally from @@IDENTITY was ' +
     CONVERT(varchar(2),@Ident)
IF (SELECT @@IDENTITY) IS NULL
   PRINT 'The value currently in @@IDENTITY is NULL'
ELSE
   PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY)
-- В следующей строке просто формируется разделитель во время печати
PRINT ''
/* Выполнение следующей строки должно окончиться неудачей, поскольку одним
** из столбцов таблицы является столбец первичного ключа, а в качестве
** первичного ключа не может быть задано NULL-значение. Переменная @@IDENTITY
** имеет NULL-значение, поскольку за несколько операторов до этого был
** выполнен оператор INSERT, а таблица, в которую выполнена вставка, не имеет
** столбца идентификации. По-видимому, наиболее важная особенность этой
** ситуации состоит в том, что значение @@IDENTITY изменилось сразу же после
** выполнения предыдущего оператора INSERT */
INSERT INTO TestChild2
VALUES
     (@@IDENTITY)
```

Описание полученных результатов

Приведенный выше сценарий предназначен для определения того, что происходит, если значение `@@IDENTITY` используется непосредственно, а не перемещается в безопасное место. После вызова на выполнение этого сценария все действия выполняются успешно вплоть до последнего оператора `INSERT`. В рассматриваемом последнем операторе предпринимается также попытка непосредственно использовать значение `@@IDENTITY`, но это значение уже изменилось в результате выполнения предыдущего оператора `INSERT`. Предыдущий оператор применяется к таблице без столбца идентификации, поэтому значение `@@IDENTITY` становится равным `NULL`. А в связи с тем. что в качестве первичного ключа нельзя применять NULL-значение, попытка выполнить последний оператор `INSERT` оканчивается неудачей:

```
(1 row(s) affected)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is 1
(1 row(s) affected)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is NULL
Msg 515, Level 16, State 2, Line 44
Cannot insert the value NULL into column 'IDcol', table 'master.dbo.
TestChild2'; column does not allow nulls. INSERT fails.
```

На этом выполнение сценария заканчивается.

Чтобы исправить эту ошибку, достаточно внести одно небольшое изменение (сохранить первоначально полученное значение `@@IDENTITY`):

```
/*****************************************
* Этот сценарий показывает, что значение   *
* идентификации после выдачи еще одного    *
* оператора INSERT теряется                *
*****************************************/
DECLARE @Ident   int -- Эта переменная предназначена для хранения информации.
                     -- На ее примере показано, как перемещать значения,
                     -- полученные из системных функций, в безопасное место
INSERT INTO TestIdent
       DEFAULT VALUES
SET @Ident = @@IDENTITY
PRINT 'The value we got originally from @@IDENTITY was ' +
       CONVERT(varchar(2),@Ident)
PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY)
/* Выполнение первого оператора INSERT с использованием значения @@IDENTITY
** оканчивается успешно. Это значение остается неизменным, поскольку между
** первоначальным оператором INSERT и текущим нет других операторов. Но
** выполнение следующего оператора INSERT становится не столь успешным */
INSERT INTO TestChild1
VALUES
       (@@IDENTITY)
PRINT 'The value we got originally from @@IDENTITY was ' +
       CONVERT(varchar(2),@Ident)
IF (SELECT @@IDENTITY) IS NULL
   PRINT 'The value currently in @@IDENTITY is NULL'
ELSE
   PRINT 'The value currently in @@IDENTITY is ' + CONVERT(varchar(2),@@IDENTITY)
-- В следующей строке просто формируется разделитель во время печати
PRINT ''

/* На этот раз вставка данных завершается успешно, поскольку используется не
** непосредственно полученное значение @@IDENTITY, а хранимое значение */
INSERT INTO TestChild2
VALUES
     (@Ident)
```

На этот раз выполнение сценария оканчивается успешно:

```
(1 row(s) affected)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is 1
(1 row(s) affected)
The value we got originally from @@IDENTITY was 1
The value currently in @@IDENTITY is NULL
(1 row(s) affected)
```

В данном примере было довольно легко определить причину возникновения проблемы, поскольку обнаруживалась попытка вставить в столбец первичного ключа NULL-значение. Но не каждый сценарий ошибочной реализации способов формирования связей между таблицами может оказаться таким наглядным. В частности, дела обстояли бы совсем иначе, если бы во второй таблице был столбец идентификации. В таком случае могло бы вполне оказаться, что в таблицу вставляют фиктивные данные, но никто об этом не догадывается, по крайней мере до тех пор, пока не возникает весьма серьезная проблема нарушения целостности данных!

Использование системной переменной @@ROWCOUNT

При выполнении многочисленных запросов, которые рассматривались до сих пор в данной книге, всегда было довольно легко узнать, какое количество строк затронуто тем или иным оператором, В частности, информация об этом могла быть получена с помощью программы Query Analyzer. Например, после вызова на выполнение оператора

```
USE Northwind

SELECT * FROM Categories
```

не только выводятся все строки таблицы `Categories`, но и отображаются данные о количестве строк, затронутых данным запросом (в этом случае таковыми являются все строки таблицы):

```
(8 row(s) affected)
```

Но иногда требуется определить количество строк, затронутых конкретным запросом, программным путем. Кроме `@@IDENTITY`, еще одним бесценным инструментальным средством, позволяющим определить, что происходит в сценарии, является системная переменная `@@ROWCOUNT`, но данная переменная возвращает значение, которое указывает количество затронутых строк, а не идентификационное значение.

Рассмотрим более подробно, как применяется эта системная переменная на практике, с помощью следующего примера:

```
USE Northwind
GO
DECLARE @RowCount int -- Обратите внимание на наличие лишь одного знака @
SELECT * FROM Categories
SELECT @RowCount = @@ROWCOUNT

PRINT 'The value of @@ROWCOUNT was ' + CAST(@RowCount AS varchar(5))
```

Этот сценарий также показывает количество возвращенных строк, но, кроме того, в полученном выводе появляется такая новая строка:

```
The value of @@ROWCOUNT was 8
```

Конкретные способы применения системной переменной `@@ROWCOUNT` будут рассматриваться при описании хранимых процедур ниже в настоящей книге. А пока что достаточно понять, что с помощью `@@ROWCOUNT` можно подробнее узнать о том, что происходит при выполнении оператора. К тому же ее применение не ограничивается операторами `SELECT`, поскольку операторы `UPDATE`, `INSERT` и `DELETE` также устанавливают значение `@@ROWCOUNT`.

> Приведенный выше пример также показывает, что и значение `@@ROWCOUNT` желательно сохранять во вспомогательной переменной, во многом аналогично тому, как сохраняется значение `@@IDENTITY`. Дело в том, что значение `@@ROWCOUNT` переустанавливается после выполнения каждого следующего оператора, поэтому, если значение `@@ROWCOUNT`, полученное на каком-то этапе, в дальнейшем будет использоваться для осуществления целого ряда действий, то его следует перенести в безопасное место.

Пакеты

Пакет – это средство группирования операторов T-SQL в виде одной логической единицы. Все операторы, содержащиеся в пакете, объединяются в один план выполнения, поэтому в процессе синтаксического анализа рассматриваются все операторы, и все они должны успешно пройти проверку синтаксиса, так как в противном случае не будет выполнен ни один из операторов. Тем не менее следует отметить, что успешное завершение проверки синтаксиса не исключает возможности возникновения ошибок на этапе прогона. В случае возникновения ошибки этапа прогона все еще сохраняются результаты всех операторов, которые были выполнены до этой ошибки. Подводя итог, можно отметить, что если синтаксический анализ одного из операторов на этапе проверки синтаксиса оканчивается неудачей, то не выполняется ни один из операторов. Если же оканчивается неудачей выполнение одного из операторов на этапе прогона, то остаются в силе результаты выполнения всех операторов, предшествующих тому, при выполнении которого возникла ошибка.

Все сценарии, рассматривавшиеся нами до сих пор, можно считать отдельными пакетами. Одним пакетом считается даже тот большой сценарий, описание которого приведено выше в данной главе. Чтобы разделить сценарий на несколько пакетов, можно воспользоваться оператором `GO`. Оператор `GO` характеризуется описанными ниже особенностями.

- Этот оператор должен находиться на отдельной строке (на той же строке не должно быть больше ничего, за исключением комментариев); из этого правила есть исключение, которое будет вскоре описано, но следует всегда исходить из того, что оператор `GO` должен находиться на отдельной строке.
- Оператор `GO` вызывает компиляцию всех операторов от начала сценария или от предыдущего оператора `GO` (в зависимости от того, что ближе), после чего полученный план выполнения передается на сервер независимо от всех других пакетов.
- Оператор `GO` – это не оператор языка T-SQL, а скорее команда, распознаваемая различными утилитами SQL Server с интерфейсом командной строки (OSQL, ISQL и Query Analyzer).

Причина, по которой оператор GO должен находиться на отдельной строке

Оператор `GO` должен находиться на отдельной строке. С формальной точки зрения определение нового пакета может начинаться на той же строке, где находится оператор `GO`, вслед за этим оператором, но в таком случае удобство чтения значительно снижается. Кроме того, перед оператором `GO` в той же строке не могут находиться другие операторы T-SQL, так как это часто приводит к получению ошибочных результатов синтаксического анализа и может либо вызвать ошибку синтаксического анализа, либо привести к возникновению других непредвиденных ситуаций. Например, синтаксический анализатор, встретив оператор `GO` после конструкции `WHERE`, как показано в следующем примере:

```
SELECT * FROM Customers WHERE CustomerID = 'ALFKI' GO
```

выводит такое сообщение об ошибке:

```
Msg 102, Level 15, State 1, Line 1
Incorrect syntax near 'GO'.
```

Отдельная отправка каждого пакета на сервер

Каждый пакет обрабатывается независимо от других пакетов, поэтому ошибка при выполнении одного пакета не исключает возможности успешного выполнения другого пакета. В качестве иллюстрации рассмотрим следующий пример кода:

```
USE AdventureWorks
DECLARE @MyVarchar varchar(50)  -- Действие этого оператора DECLARE
                                -- распространяется только на данный пакет!
SELECT @MyVarchar = 'Honey, I''m home...'
PRINT 'Done with first Batch...'
GO
PRINT @MyVarchar  -- При выполнении этого оператора возникает ошибка, поскольку
                  -- переменная @MyVarchar не объявлена в данном пакете
PRINT 'Done with second Batch'
GO
PRINT 'Done with third batch' -- Следует отметить, что этот пакет выполняется,
                              -- несмотря на то, что перед этим возникла ошибка

GO
```

Если бы между пакетами, содержащимися в этом коде, имелись зависимости, то происходило бы аварийное завершение сразу всех пакетов из-за ошибки во втором пакете или по крайней мере окончилось бы неудачей выполнение и этого пакета, и всех следующих, но этого не происходит, как показывают следующие результаты выполнения приведенного выше сценария:

```
Done with first Batch...
Msg 137, Level 15, State 2, Line 2
Must declare the scalar variable "@MyVarchar".
Done with third batch
```

Эти результаты подтверждают, что каждый пакет является полностью автономным с точки зрения влияния на него нарушений в работе, возникающих на этапе прогона. Но следует помнить, что зависимости между пакетами могут возникать исходя из того, что в одном пакете будет предприниматься попытка выполнить работу, зависящую от результатов успешного выполнения предыдущего пакета. Дополнительная информация на эту тему приведена в следующем разделе, в котором речь идет о том, какая информация передается и не передается из одного пакета в другой.

GO как команда, а не оператор языка T-SQL

Одной из распространенных ошибок является мнение о том, что GO — оператор языка T-SQL. В действительности GO — это команда, которая распознается только инструментальными средствами редактирования (Management Studio — SQLCMD). Некоторые инструментальные средства независимых разработчиков не поддерживают команду GO, но большинство тех средств, которые предназначены для использования в СУБД SQL Server, обеспечивают такую поддержку.

Инструментальное средство редактирования, обнаружив оператор GO, распознает его как флажок, обозначающий конец пакета, оформляет этот пакет и отправляет его в виде отдельной единицы работы на сервер, предварительно исключив оператор GO. Необходимость в выполнении последнего действия обусловлена тем, что в сервере полностью отсутствует какая-либо поддержка оператора GO.

При попытке включить оператор GO в транзитный запрос с использованием ODBC, OLE DB, ADO, ADO.NET или любого другого метода доступа сервер возвращает сообщение об ошибке. Оператор GO воспринимается только инструментальными средствами и служит индикатором того, что заканчивается текущий пакет и начинается другой.

Ошибки в пакетах

Ошибки в пакетах подразделяются на две категории:

- синтаксические ошибки;
- ошибки, обнаруживаемые на этапе прогона программы.

Если синтаксический анализатор запросов обнаруживает в пакете **синтаксическую ошибку**, обработка такого пакета немедленно прекращается. Проверка синтаксиса осуществляется перед компиляцией и выполнением пакета, поэтому неудачное завершение проверки синтаксиса приводит к тому, что не выполняется ни одна часть пакета, независимо от того, в каком месте пакета была обнаружена синтаксическая ошибка.

Ошибки этапа прогона программы проявляются совсем иначе. Ко времени возникновения ошибки этапа прогона выполнение всех предшествующих этому моменту операторов заканчивается, поэтому все последствия выполнения этих операторов остаются неизменными, если сами эти операторы не входят в состав незафиксированной транзакции. (Транзакции рассматриваются в главе 14, но о них необходимо упомянуть и в данном контексте, поскольку транзакции выполняются как одна единица работы, а в случае неудачного завершения происходит полная отмена внесенных ими изменений.) Действия, осуществляемые вслед за ошибкой этапа прогона программы, зависят от характера ошибки. Вообще говоря, обнаружение ошибок этапа прогона программы приводит к прекращению выполнения пакета, начиная с того момента, в который была обнаружена ошибка, т.е. оставшаяся часть пакета не выполняется. Но некоторые ошибки этапа прогона программы, такие как нарушения ссылочной целостности, препятствуют лишь выполнению оператора, содержащего ошибку, а все прочие операторы в пакете все равно выполняются. В связи с возможностью такого сценария развития событий становится понятным, почему так важно обеспечить проверку наличия ошибок; более подробно тема проверки наличия ошибок будет раскрыта в главе, посвященной хранимым процедурам (глава 12).

Рекомендации по использованию пакетов

Пакеты имеют несколько назначений, но все области их применения имеют общую особенность — пакеты используются для выполнения каких-либо действий, которые должны быть осуществлены либо заблаговременно, либо отдельно от всех прочих действий в сценарии.

Операторы, которые должны выполняться в составе отдельных пакетов

Некоторые операторы, безусловно, необходимо включать в состав отдельных пакетов. К ним относятся следующие операторы:

- CREATE DEFAULT;
- CREATE PROCEDURE;
- CREATE RULE;
- CREATE TRIGGER;
- CREATE VIEW.

Если какой-либо из этих операторов желательно включить совместно с другими операторами в один сценарий, то следует выделить их в составе сценария в отдельный пакет с помощью оператора GO.

> Следует отметить, что операторы удаления объектов, DROP, также желательно помещать в отдельный пакет или по крайней мере включать в один пакет с другими операторами DROP. Дело в том, что если в дальнейшем в том же сценарии должен быть создан объект с тем же именем, что и у удаляемого объекта, то синтаксический анализ оператора CREATE в процессе обработки пакета окончится неудачей, при условии, что к этому времени еще не произошло уничтожение объекта с тем же именем, которое указано в операторе CREATE. Это означает, что оператор DROP должен быть выполнен в составе отдельного, предшествующего пакета, чтобы ко времени вызова пакета с оператором CREATE уже произошло удаление объекта.

Использование пакетов для определения порядка следования заданий

По-видимому, чаще всего пакеты применяются в таких ситуациях, когда необходимо регламентировать порядок следования заданий, иными словами, когда требуется обеспечить, чтобы одно задание полностью завершалось перед запуском следующего задания. Чаще всего СУБД SQL Server вполне успешно справляется с этой ситуацией: вначале выполняется первый оператор в сценарии, после чего можно вполне рассчитывать на то, что ко времени выполнения второго оператора сценария сервер будет находиться в должном состоянии. Однако в некоторых ситуациях СУБД SQL Server не может успешно справиться с проблемой управления порядком следования операторов.

Рассмотрим пример, в котором создается база данных вместе с некоторыми таблицами:

```
CREATE DATABASE Test
CREATE TABLE TestTable
(
  col1   int,
  col2   int
)
```

После вызова этого сценария на выполнение на первый взгляд создается такое впечатление, что все действия завершены успешно:

```
Command(s) completed successfully.
```

В действительности же дело обстоит иначе — достаточно проверить схему INFORMATION_SCHEMA в базе данных Test, в результате чего обнаруживается, что некоторые объекты в этой базе данных отсутствуют:

```
SELECT TABLE_CATALOG FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME =
  'TestTable'
TABLE_CATALOG
--------------------------------------------------------------------------
-------------------------
master

(1 row(s) affected)
```

Как оказалось, таблица создана не в той базе данных, в которой требовалось. Это связано с тем, что ко времени выполнения оператора CREATE TABLE текущей была не требуемая, а другая база данных. В рассматриваемом случае оказалось, что таковой является база данных master, поэтому таблица была создана именно в ней.

Следует отметить, что при проверке этого примера в других обстоятельствах текущей могла бы оказаться база данных, отличная от master, поэтому были бы получены другие результаты. Но именно в этом заключается самое важное – текущей может оказаться практически любая база данных. С этим и связана неоднократно подчеркиваемая важность использования оператора USE.

Даже поверхностные размышления на эту тему могут привести к заключению, что возникшую ошибку легко исправить, — достаточно предусмотреть применение оператора USE, но прежде чем проверить это предположение, необходимо удалить старую (вернее, не такую уж старую) базу данных:

```
USE MASTER
DROP DATABASE Test
```

Теперь попытаемся вызвать на выполнение сценарий с внесенными в него изменениями:

```
CREATE DATABASE Test

USE Test

CREATE TABLE TestTable
(
  col1  int,
  col2  int
)
```

К сожалению, при выполнении этого сценария возникают другие проблемы:

```
Msg 911, Level 16, State 1, Line 3
Could not locate entry in sysdatabases for database 'Test'. No entry found
with that name. Make sure that the name is entered correctly.
```

Синтаксический анализатор при проверке кода сценария обнаруживает, что в операторе USE упоминается база данных, которая к этому моменту еще не существует. Такая ситуация как раз и демонстрирует, для чего нужны пакеты. Дело в том, что обработка оператора CREATE DATABASE должна быть завершена к тому моменту, когда будет предпринята попытка использовать новую базу данных:

```
CREATE DATABASE Test
GO

USE Test
CREATE TABLE TestTable
(
  col1  int,
  col2  int
)
```

Такая организация работы намного лучше. Безусловно, это не обнаруживается в непосредственно полученных результатах:

```
Command(s) completed successfully.
```

Но после вызова на выполнение того же запроса к схеме INFORMATION_SCHEMA подтверждается, что результаты оказались успешными:

```
TABLE_CATALOG
------------------------------------------------------------------------------
--------
Test
(1 row(s) affected)
```

Рассмотрим еще один пример, который даже более явно показывает необходимость управлять порядком следования заданий.

Если с помощью оператора ALTER TABLE вносятся существенные изменения в типы столбцов или добавляются новые столбцы. результатами этих изменений нельзя воспользоваться до тех пор, пока не завершится выполнение пакета, в котором вносятся такие изменения.

Предпримем попытку ввести дополнительный столбец в таблицу TestTable базы данных Test, а затем попытаемся обратиться к этому столбцу в ходе выполнения одного и того же пакета следующим образом:

```
USE Test
ALTER TABLE TestTable
  ADD col3 int
INSERT INTO TestTable
(col1, col2, col3)
VALUES
(1,1,1)
```

В результате будет получено следующее сообщение об ошибке, обусловленное тем, что СУБД SQL Server не может найти в своих справочных таблицах информацию о новом столбце с указанным именем:

```
Msg 207, Level 16, State 1, Line 6
Invalid column name 'col3'.
```

Но достаточно ввести один простой оператор GO вслед за строкой ADD col3 int, и выполнение сценария завершится успешно:

```
(1 row(s) affected)
```

Утилита SQLCMD

SQLCMD – это утилита, которая позволяет вызывать на выполнение сценарии из приглашения к вводу команд командного интерпретатора Windows. Такой способ вызова может оказаться очень удобным для выполнения сценариев преобразования данных или сопровождения базы данных, а также может служить быстрым и легким способом перехвата вывода сценариев в виде текстовых файлов.

Утилита SQLCMD пришла на замену применявшейся ранее программе OSQL. Программа OSQL все еще входит в состав программного обеспечения SQL Server, но лишь для обратной совместимости. А утилита ISQL с интерфейсом командной строки, относящаяся к еще более ранним выпускам, больше не поддерживается.

Синтаксис оператора вызова утилиты SQLCMD на выполнение из командной строки включает большое количество различных параметров и выглядит следующим образом:

```
sqlcmd
[
{ { -U <login id> [ -P <password> ] } |  - E }
]
[-S <server name> [ \<instance name> ] ] [ -H <workstation name> ] [ -d <db name> ]
[ -l <time out> ] [ -t <time out> ] [ -h <headers> ]
[ -s <col separator> ] [ -w <col width> ] [ -a <packet size> ]
[ -e ] [ -I ]
[ -c <cmd end> ] [ -L [ c ] ] [ -q "<query>" ] [ -Q "<query>" ]
[ -m <error level> ] [ -V ] [ -W ] [ -u ] [ -r [ 0 | 1 ] ]
[ -i <input file> ] [ -o <output file> ]
[ -f <codepage> | i:<codepage> [ <, o: <codepage> ]
[ -k [ 1 | 2 ] ]
[ -y <display width> ] [-Y <display width> ]
[ -p [ 1 ] ] [ -R ] [ -b ] [ -v ] [ -A ] [ -X [ 1 ] ] [ -x ]
[ -? ]
]
```

Изучая эти параметры, необходимо учитывать одну из наиболее важных их особенностей, состоящую в том, что большинство этих параметров (но, как ни странно, не все) чувствительны к регистру. Например, и параметр "`-Q`", и параметр "`-q`" указывает, что должен быть выполнен запрос, но первый говорит о том, что нужно выйти из программы SQLCMD по окончании выполнения запроса, а второй – что этого не нужно делать.

Итак, попытаемся вызвать на выполнение короткий запрос непосредственно из командной строки, как показано ниже. Еще раз напомним, что этот запрос должен быть вызван на выполнение из командной строки интерпретатора команд Windows (не используйте программу Management Console).

```
SQLCMD -Usa -Pmypass -Q "SELECT * FROM Northwind..Shippers"
```

Параметр `-P` служит для указания пароля. Если настройка конфигурации сервера выполнена с учетом использования непустого пароля (а так и должно быть!), то пароль доступа к базе данных должен быть указан непосредственно вслед за параметром `-P`, без пробела между ними.

После вызова этого запроса на выполнение из приглашения к вводу команд должны быть получены примерно такие результаты:

```
C:\>Osql -Usa -Pmypass -Q "SELECT * FROM Northwind..Shippers"
ShipperID      CompanyName          Phone
----------     -----------------    --------------------
1              Speedy Express       (503) 555-9831
2              United Package       (503) 555-3199
3              Federal Shipping     (503) 555-9931
(3 rows affected)
C:\>
```

Теперь рассмотрим короткий пример создания текстового файла, чтобы показать, как осуществляется запись результатов запроса в файл. В приглашении к вводу команд введите следующее:

```
C:\>copy con testsql.sql
```

После ввода этой команды курсор должен перейти на новую, пустую строку (без какого-либо приглашения); введите в этой строке следующую команду:

```
SELECT * FROM Northwind..Shippers
```

Затем последовательно нажмите клавиши <F6> и <Return> (на этом создание нового текстового файла заканчивается). После этого должно появиться такое сообщение:

```
1 file(s) copied.
```

Теперь попытаемся вызвать на выполнение тот же запрос, что и раньше, но с использованием файла сценария с заданным для него именем. При этом команда в приглашении к вводу команд немного изменяется:

```
C:\>sqlcmd -Usa -Pmypass -i testsql.sql
```

Выполнение этой команды должно привести точно к таким же результатам, как и после выполнения запроса с параметром -Q. Но основное отличие, безусловно, состоит в том, что команда была вызвана на выполнение из файла. Удобство применения файлов состоит в том, что файлы могут содержать сотни (или даже тысячи) различных команд.

Практическое занятие

Применение утилиты SQLCMD

В последнем примере применения утилиты SQLCMD, рассматриваемом в данной главе, воспользуемся этой утилитой для формирования текстового файла, который можно было бы импортировать в другое приложение для анализа (например, в приложение Excel).

Напомним, что в главе 10 рассматривалось представление, которое позволяло получить список заказов за вчерашний день. Вначале откроем текст базового запроса этого представления и введем данный текст в текстовый файл:

```
C:\>copy con YesterdaysOrders.sql
```

После выполнения этой команды курсор должен снова перейти на пустую строку (без какого-либо приглашения), в которой можно ввести следующий запрос:

```
SELECT cu.CompanyName,
       o.OrderID,
       o.OrderDate,
       od.ProductID,
       p.ProductName,
       od.Quantity,
       od.UnitPrice,
       od.Quantity * od.UnitPrice AS ExtendedPrice
FROM   Customers AS cu
INNER JOIN  Orders AS o
     ON cu.CustomerID = o.CustomerID
INNER JOIN  [Order Details] AS od
     ON o.OrderID = od.OrderID
INNER JOIN  Products AS p
     ON od.ProductID = p.ProductID
WHERE CONVERT(varchar(12),o.OrderDate,101) =
     CONVERT(varchar(12),DATEADD(day,-1,GETDATE()),101)
```

После этого снова нажмите клавиши <F6> и <Return>, чтобы указать операционной системе Windows, что ввод текста файла закончен и его необходимо закрыть.

Итак, текстовый файл с исходным кодом запроса уже готов и почти полностью сделано все, что необходимо для получения выходных данных с помощью утилиты SQLCMD. Но вначале требуется ввести некоторые данные, относящиеся ко вчерашнему дню. (В данных, используемых при выполнении рассматриваемого примера, не должно быть сведений о заказах, относящихся ко вчерашнему дню, если только вы уже не выполнили этот сценарий и не ввели некоторые заказы. Подчеркнем, что речь идет о том сценарии ввода заказов, который был приведен в главе о представлениях и еще раз откорректирован в настоящей главе.) Итак, учитывая сказанное, еще раз вызовем на выполнение следующий сценарий ввода заказов (при желании для этого можно воспользоваться окном Query):

```
USE Northwind
DECLARE @Ident int
INSERT INTO Orders
(CustomerID,OrderDate)
VALUES
('ALFKI', DATEADD(day,-1,GETDATE()))
SELECT @Ident = @@IDENTITY
INSERT INTO [Order Details]
(OrderID, ProductID, UnitPrice, Quantity)
VALUES
(@Ident, 1, 50, 25)

SELECT 'The OrderID of the INSERTed row is ' + CONVERT(varchar(8),@Ident)
```

После успешного выполнения этого сценария в базе данных должна быть по меньшей мере одна строка с информацией о заказе, которая относится ко вчерашнему дню. Таким образом, проведена почти вся необходимая подготовка, но, как было сказано, мы поставили перед собой задачу записать результаты в текстовый файл, поэтому на сей раз должны ввести в команду вызова утилиты SQLCMD некоторые дополнительные параметры, чтобы сообщить СУБД SQL Server, в какой файл должны быть записаны выходные данные:

```
C:\Documents and Settings\robv.BARNICLE>
  sqlcmd -UMyLogin -PMyPass -iYesterdaysOrders.sql -oYesterdaysOrders.txt
```

После выполнения этой команды утилита SQLCMD не выводит каких-либо особых сообщений, а просто снова появляется приглашение к вводу команд командного интерпретатора Windows, указывающее на то, какое дисковое устройство является текущим (наиболее вероятно, `C:\`). Теперь мы можем проверить, что находится в файле `YesterdaysOrders.txt`, следующим образом:

```
C:\>TYPE YesterdaysOrders.txt
```

В результате этого на экран будет выведено содержимое файла с выходными данными, которые свидетельствуют о наличии в базе данных информации об одном заказе за вчерашний день:

```
Changed database context to 'Northwind'.
CompanyName        OrderID    OrderDate               ProductID   ProductName
Quantity UnitPrice
 ExtendedPrice
---------------------------------------- ----------- ----------------------
-- ---
-------- ---------------------------------------- -------- ---------------
------
 ----------------------
Alfreds Futterkiste      11078 2005-08-27 14:35:50.163       1   Chai
25   50.0000
          1250.0000

(1 rows affected)
```

Описание полученных результатов

Прежде всего, необходимые операторы SQL были объединены в один сценарий, который начинается с оператора `USE`, а за ним следует оператор `SELECT`, выполняющий необходимые действия.

После этого оператор выборки был вызван на выполнение в составе сценария с помощью утилиты SQLCMD. Как и в предыдущих примерах данной главы, с помощью параметров `-U` и `-P` была задана информация о регистрационном имени пользователя и пароле. Параметр `-i` указывает утилите SQLCMD, что ввод операторов должен осуществляться из входного файла, а непосредственно за обозначением параметра `-i` было приведено имя этого файла. Наконец, включен параметр `-o`, который указывает утилите SQLCMD, что вывод сценария должен быть записан в файл (и, безусловно, вслед за этим параметром приведено имя файла — `YesterdaysOrders.txt`). То, что оба эти файла имеют одинаковое имя, `YesterdaysOrders`, не должно вызвать путаницу, поскольку это — два отдельных файла, имеющих разные расширения, `.sql` и `.txt`, которые подчеркивают их различное назначение.

Количество различных параметров утилиты SQLCMD достаточно велико, но наиболее важными из них являются параметры с обозначением регистрационного имени и пароля, а также параметр, который сообщает, как оформлено задание для утилиты (в виде непосредственно заданного запроса или входного файла). Хотя на первый взгляд возможности этого инструментального средства с интерфейсом командной строки кажутся ограниченными, оно позволяет применять различные варианты и сочетания параметров и тем самым реализовывать довольно сложные варианты обработки данных.

Динамический код SQL. Формирование кода в оперативном режиме с помощью команды EXEC

Безусловно, возможность сохранять код сценариев в файлах является очень удобной, но иногда вплоть до этапа прогона остается неизвестным, какой код должен быть выполнен.

Кстати, отметим, что в настоящей главе утилита SQLCMD больше не используется; следующие примеры должны выполняться с помощью программы Management Console.

С учетом определенных нюансов СУБД SQL Server позволяет оперативно формировать операторы SQL, используя средства манипулирования строками. Необходимость в этом обычно возникает в связи с тем, что некоторые сведения, требуемые для выполнения операторов, остаются неизвестными вплоть до этапа прогона. Синтаксическая структура операторов динамического формирования кода выглядит примерно таким образом:

```
EXEC ({<string variable>|'<literal command string>'})
```

или таким:

```
EXECUTE ({<string variable>|'<literal command string>'})
```

Как и при вызове на выполнение хранимых процедур, допускается использовать и то и другое ключевое слово (EXEC и EXECUTE).

Рассмотрим применение динамических запросов на примере базы данных Northwind. Для этого создадим фиктивную таблицу, из которой будет осуществляться динамическая выборка информации:

```
USE Northwind
GO
-- Создание таблицы, в которой будет представлена информация
-- для динамического кода SQL
CREATE TABLE DynamicSQLExample
(
  TableID   int   IDENTITY   NOT NULL
    CONSTRAINT PKDynamicSQLExample
              PRIMARY KEY,
  TableName varchar(128)     NOT NULL
)
GO
/* Заполнение таблицы. На этот раз используются данные обо всех
** пользовательских таблицах в базе данных */
INSERT INTO DynamicSQLExample
SELECT TABLE_NAME
  FROM Information_Schema.Tables
  WHERE TABLE_TYPE = 'BASE TABLE'
```

Выполнение этого сценария должно привести к получению примерно таких результатов:

```
(17 row(s) affected)
```

Как уже было сказано, результаты, полученные читателем, могут оказаться другими. Это зависит от того, какие примеры из книги были вами выполнены, какие вы пропустили и в отношении каких таблиц вы проявили инициативу, уничтожив следы выполнения упражнений после их завершения. Так или иначе, вышеназванные различия не имеют особого значения.

Итак, на данный момент сформирован список всех таблиц в текущей базе данных. Теперь предположим, что требуется обеспечить выборку некоторых данных, хранящихся в одной из таблиц, но таблица должна быть указана только на этапе прогона с помощью ее идентификатора. Например, выборка всех данных из таблицы с идентификатором 14 может быть выполнена следующим образом:

```
/* Вначале объявляется переменная для хранения имени таблицы.
** Следует учитывать, что имена объектов могут иметь длину 128 символов */
DECLARE @TableName     varchar(128)

-- Определить имя таблицы, имеющей указанный идентификатор
SELECT @TableName = TableName
   FROM DynamicSQLExample
   WHERE TableID = 14
-- Наконец, это значение передается в оператор EXEC
EXEC ('SELECT * FROM ' + @TableName)
```

Если имена таблиц, собранные в таблице DynamicSQLExample при выполнении этого упражнения читателем, совпадают с именами, полученными автором, то идентификатор TableID, равный 14, должен соответствовать таблице Categories. В этом случае должен быть получен примерно такой результат, как показано в табл. 11.2 (в целях сокращения объема таблицы правые столбцы были исключены).

Таблица 11.2. Часть данных, хранящихся в таблице Categories

Столбец CategoryID	Столбец CategoryName	Столбец Description	...
1	Beverages	Soft drinks, coffees, teas, beers, and ales	...
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings	...
3	Confections	Desserts, candies, and sweet breads	...
4	Dairy Products	Cheeses	...
5	Grains/Cereals	Breads, crackers, pasta, and cereal	...
6	Meat/Poultry	Prepared meats	...
7	Produce	Dried fruit and bean curd	...
8	Seafood	Seaweed and fish	...

Нюансы, связанные с использованием оператора EXEC

Оператор EXEC относится к числу привлекательных и перспективных средств, но при его использовании обычно приходится проводить небольшие эксперименты и преодолевать определенные сложности. Ниже описаны основные нюансы, с которыми приходится сталкиваться при применении оператора EXEC.

- Выполнение оператора EXEC происходит в отдельном домене по сравнению с кодом, в котором он вызывается, иными словами, в вызывающем коде нельзя ссылаться на переменные, заданные в операторе EXEC, а в операторе EXEC не могут использоваться ссылки на переменные в вызывающем коде после того, как будет выполнена подстановка значений этих переменных в строку, передаваемую для оператора EXEC.
- Оператор EXEC выполняется в том же контексте защиты, в котором работает текущий пользователь, а не в контексте защиты вызывающего объекта.
- Оператор EXEC выполняется в том же контексте соединения и транзакционном контексте, что и вызывающий объект (дополнительная информация на эту тему приведена в главе 14).
- Операции конкатенации, для которых требуется вызов функции, должны быть выполнены в строке EXEC до фактического вызова оператора EXEC, поскольку операция конкатенации с помощью функции не может быть выполнена в том же операторе, в котором вызывается EXEC.
- Оператор EXEC не может использоваться в функциях, определяемых пользователем.

Каждое из этих требований довольно сложно понять без дополнительного пояснения, поэтому рассмотрим каждое из них отдельно.

Область определения переменных в операторе EXEC

Принципы, на основании которых устанавливается область определения переменных в операторе EXEC, с трудом поддаются пониманию. Но фактически строка с оператором, в которой происходит вызов оператора EXEC, имеет ту же область определения, что и остальная часть пакета (или процедуры), в котором выполняется оператор EXEC, но код, вызываемый на выполнение в результате применения оператора EXEC, рассматривается как принадлежащий к отдельному пакету. Поскольку такая ситуация встречается часто, проще всего показать ее на примере:

```
USE Northwind
/* Вначале объявляются переменные. Одна из них предназначена для хранения данных,
** передаваемых в оператор EXEC, а другая - для получения возвращаемых данных
** (но в данном случае возврат не происходит) */
DECLARE @InVar  varchar(50)
DECLARE @OutVar  varchar(50)
-- Подготовка строки, предназначенной для передачи в команду EXEC
SET @InVar = 'SELECT @OutVar = FirstName FROM Employees WHERE EmployeeID = 1'
-- Выполнение команды
EXEC (@Invar)
-- Теперь, чтобы убедиться в том. что результат от этого не зависит, вызовем
-- на выполнение оператор SELECT непосредственно, без использования
-- переменной @InVar
EXEC ('SELECT @OutVar = FirstName FROM Employees WHERE EmployeeID = 1')
-- Переменная @OutVar будет по-прежнему иметь NULL-значение, поскольку в нее
-- не удается поместить какие-либо данные
SELECT @OutVar
```

Теперь рассмотрим полученные результаты:

```
Msg 137, Level 15, State 1, Line 13
Must declare the scalar variable '@OutVar'.
Msg 137, Level 15, State 1, Line 16
Must declare the scalar variable '@OutVar'.
-----------------------------------------------
NULL
(1 row(s) affected)
```

Очевидно, что СУБД SQL Server сразу же сообщает об ошибке и не может предпринять каких-либо дальнейших действий. Но следует пояснить, почему получено сообщение об ошибке, указывающее на отсутствие объявления, "`Must Declare`", тогда как переменная `@OutVar` уже объявлена. Безусловно, эта переменная объявлена во внешней области определения, но не в самом операторе `EXEC`.

Рассмотрим, что произойдет, если тот же сценарий будет организован несколько иначе:

```
USE Northwind
-- На этот раз требуется только одна переменная, но потребность в ней
-- сохраняется в течение более продолжительного времени
DECLARE @InVar   varchar(200)
/* Подготовка строки для передачи в команду EXEC. На этот раз предпринимается
** попытка передать одновременно несколько операторов. Все эти операторы
** выполняются в виде одного пакета */
SET @InVar = 'DECLARE @OutVar varchar(50)
           SELECT @OutVar = FirstName FROM Employees WHERE EmployeeID = 1
           SELECT ''The Value Is '' + @OutVar'
-- Выполнение команды
EXEC (@Invar)
```

На этот раз полученные результаты больше напоминают ожидаемые:

```
------------------------------------------------------------------
The Value Is Nancy
```

Обратите внимание на то, как используются два подряд идущих символа кавычек для указания на то, что они фактически не отмечают конец строки, а служат для обозначения символа кавычки.

Итак, приведенный пример показывает, что используются две разные области определения переменных, связь между которыми отсутствует. К сожалению, отсутствует также способ передачи информации между внутренней и внешней областями определения без использования внешнего механизма, такого как временная таблица. Но следует учитывать, что если будет решено применить временную таблицу для обеспечения связи между областями определения, то любая временная таблица, созданная в области определения оператора `EXEC`, будет существовать только на протяжении времени выполнения этого оператора `EXEC`.

> С данной особенностью временных таблиц, существующих лишь в течение времени выполнения процедуры `EXEC`, нам придется снова столкнуться, когда речь пойдет о триггерах и хранимых процедурах.

Небольшое исключение из правила

Тем не менее в области определения оператора EXEC происходят определенные изменения, которые сохраняются после завершения его выполнения. Дело в том, что установленные при этом значения системных переменных сохраняются, поэтому остается возможность использовать такие системные переменные, как @@ROWCOUNT. Рассмотрим еще один небольшой пример:

```
USE Northwind
EXEC('SELECT * FROM Customers')
SELECT 'The Rowcount is ' + CAST(@@ROWCOUNT as varchar)
```

Выполнение этого примера приводит к получению следующей строки (после вывода результирующего набора):

```
The Rowcount is 91
```

Контексты защиты и оператор EXEC

На данном этапе полностью раскрыть эту тему достаточно сложно, поскольку в этой и предыдущей главах еще не рассматривались хранимые процедуры и средства защиты. Тем не менее описанию оператора EXEC посвящена эта глава, а не глава, относящаяся к хранимым процедурам, поэтому дадим предварительное описание данной темы (обсуждение, начатое в настоящей главе, будет продолжено в главе, посвященной хранимым процедурам, и читателю следует об этом помнить).

Из того, что некоторому пользователю предоставляется право вызывать на выполнение хранимую процедуру, следует также, что этот пользователь получает право осуществлять действия, предусмотренные в хранимой процедуре. Например, предположим, что в базе данных имеется хранимая процедура, позволяющая получить списки всех служащих, принятых на работу в течение последнего года. Любой, кто имеет право вызывать эту хранимую процедуру на выполнение, может это сделать (и получить результаты), даже не имея права обращаться непосредственно к таблице Employees. Это действительно удобно по причинам, которые будут описаны более подробно в главе, посвященной хранимым процедурам.

Многие разработчики полагают, что те же принципы, которые определяют возможности, обусловленные предоставлением прав, распространяются и на операторы EXEC, но дело обстоит иначе. Любая ссылка на объект, применяемая в операторе EXEC, должна быть раскрыта в контексте защиты текущего пользователя. Таким образом, предположим, что некоторому пользователю предоставлено право вызывать на выполнение процедуру spNewEmployees и получать тем самым данные о служащих, принятых на работу, но у него нет прав доступа к таблице Employees. Если получение результатов в процедуре spNewEmployees обеспечивается путем выполнения простого оператора SELECT, то проблемы не возникают. А если в процедуре spNewEmployees для выполнения оператора SELECT используется оператор EXEC, то попытка выполнить оператор EXEC окончится неудачей, так как пользователь не имеет права применять оператор SELECT для доступа к таблице Employees.

Поскольку в этой и предыдущей главах еще не приведено достаточно информации о хранимых процедурах, оставим пока дальнейшее обсуждение этой темы и вернемся к ней позже в контексте описания хранимых процедур.

Использование функций для конкатенации строк, передаваемых в оператор EXEC

Фактически указанная проблема, касающаяся запрещения использования функций для конкатенации строк непосредственно в вызове оператора EXEC, может быть решена проще всего, поскольку для этого можно применить достаточно несложное решение. Указанную проблему можно сформулировать иными словами так, что не допускается вызывать какую-либо функцию для обработки строки, указанной в качестве параметра оператора EXEC, например, как показано в следующем коде:

```
USE Northwind
-- Это - неработоспособный сценарий
DECLARE @NumberOfLetters int
SET @NumberOfLetters = 15
EXEC('SELECT LEFT(CompanyName,' + CAST(@NumberOfLetters AS varchar) + ') AS
ShortName
FROM Customers')
GO
-- А этот сценарий вполне работоспособен
DECLARE @NumberOfLetters AS int
SET @NumberOfLetters = 15
DECLARE @str AS varchar(255)
SET @str = 'SELECT LEFT(CompanyName,' + CAST(@NumberOfLetters AS varchar) + ') AS
ShortName FROM Customers'
EXEC(@str)
```

При обработке первого вызова оператора EXEC формируется следующее сообщение об ошибке, поскольку подстановка всех параметров функции CAST должна быть полностью закончена еще до включения вызова этой функции в оператор EXEC:

```
Msg 102, Level 15, State 1, Line 6
Incorrect syntax near 'CAST'.
```

Но выполнение второго вызова оператора EXEC завершается успешно, поскольку в этот оператор передается уже полностью сформированная строка:

```
ShortName
---------------
Alfreds Futterk
Ana Trujillo Em
...
...
Wolski  Zajazd
```

Оператор EXEC и пользовательские функции

Эта тема не может быть раскрыта на данном этапе, поскольку в настоящей и предыдущей главах описание пользовательских функций еще не приведено, но достаточно отметить, что не допускается использовать оператор EXEC для вызова на выполнение динамического кода SQL в пользовательской функции. (Тем не менее применение операторов EXEC в хранимых процедурах в некоторых случаях вполне допустимо.)

Резюме

Для успешного освоения всей тематики программирования приложений для СУБД SQL Server очень важно достичь полного понимания того, в чем состоит назначение сценариев и пакетов. Дело в том, что сценарии и пакеты широко применяются для осуществления чрезвычайно широкого спектра функций, начиная с создания полноценной базы данных и заканчивая программированием хранимых процедур и триггеров.

Область определения локальных переменных ограничивается только одним пакетом. Предположим, что некоторая переменная уже объявлена в каком-то пакете, который относится к общему сценарию, состоящему из нескольких пакетов. Прежде чем сослаться на переменную с тем же именем в новом пакете, необходимо еще раз объявить эту переменную (и присвоить ей значение), так как в противном случае будет сформировано сообщение об ошибке.

В программном обеспечении SQL Server предусмотрено больше тридцати системных переменных. В настоящей главе описаны наиболее широко используемые системные переменные, но на этом информация по указанной теме далеко не исчерпывается. Чтобы больше узнать об этих и других системных переменных, ознакомьтесь с документацией Books Online или прочитайте приложение Б к настоящей книге. Системные переменные не требуют объявления и всегда являются доступными. Область действия некоторых системных переменных распространяется на весь сервер, тогда как значения других системных переменных относятся только к текущему соединению.

Пакеты позволяют регламентировать порядок выполнения различных частей сценария. Выполнение первого пакета начинается с началом сценария и заканчивается после обнаружения конца сценария или первого оператора `GO` (в зависимости от того, какое из этих событий возникнет раньше). Запуск следующего пакета (если таковой имеется) происходит с той строки, которая следует за концом первого пакета (после оператора `GO`), а его выполнение продолжается до обнаружения конца сценария или следующего оператора `GO` (опять-таки в зависимости от того, что будет обнаружено раньше). Такой процесс запуска пакетов продолжается до конца сценария. Итак, первый пакет, считая от начала сценария, выполняется в первую очередь, вслед за ним выполняется второй пакет и т.д. Все операторы каждого пакета должны успешно пройти проверку в синтаксическом анализаторе запросов, так как в противном случае пакет не будет передан на выполнение; однако синтаксический анализ всех прочих пакетов осуществляется отдельно, после чего происходит их запуск (если они успешно проходят этап синтаксического анализа).

Наконец, в этой главе описано, как можно создавать код SQL и вызывать его на выполнение динамически. Благодаря этому появляется возможность находить способы решения задач, в которых невозможно с полной достоверностью определить заранее состав используемого сценария, или таких задач, в которых структура применяемых операторов должна фактически зависеть от содержания отдельных фрагментов данных.

В следующих нескольких главах показано, как применяются принципы формирования сценариев и пакетов для создания хранимых процедур и триггеров — объектов СУБД SQL Server, которые в наибольшей степени напоминают настоящие программы.

Упражнения

11.1. Напишите простой сценарий, в котором создаются две целочисленные переменные (с именами `Var1` и `Var2`) со значениями `2` и `4`, а затем осуществляется суммирование и вывод суммы значений этих двух переменных.

11.2. Создайте переменную `MinOrder` и присвойте ей значение, равное минимальной стоимости товарной позиции в заказе заказчика с идентификатором `CustomerNo`, равным `'ALFKI'`, хранящемся в базе данных `Northwind`. (*Предостережение.* В данном случае речь идет об обработке денежных сумм, поэтому не следует использовать для переменной `MinOrder` тип данных `int`.) После этого выведите окончательное значение `MinOrder`.

11.3. Выведите результаты выполнения запроса `SELECT COUNT(*) FROM Customers` в окно терминала с использованием утилиты SQLCMD.

12

Хранимые процедуры

Отличительная особенность хранимых процедур состоит в том, что в них широко используются программные средства, которые аналогичны применяемым во многих процедурных языках. Хранимые процедуры обычно составляют значительную часть программного обеспечения SQL Server, а начиная с версии SQL Server 2005 обеспечивается взаимодействие хранимых процедур с инфраструктурой .NET, благодаря чему предоставляемые ими возможности еще больше расширяются.

Хранимая процедура представляет собой оформленный особым образом сценарий (вернее, пакет), который хранится в базе данных, а не в отдельном файле. Хранимые процедуры отличаются от сценариев тем, что в них допускается использование входных и выходных параметров, а также возвращаемых значений, которые фактически не могут использоваться в обычном сценарии.

Основным языком программирования для СУБД SQL Server продолжает оставаться T-SQL, который не обеспечивает такую же поддержку разносторонних способов управления ходом выполнения программы, как процедурные языки наподобие C, C++, Visual Basic, Java, Delphi и т.д. Однако T-SQL становится буквально непревзойденным, когда с его помощью осуществляются операции определения данных, манипулирования данными и доступа к данным, а предусмотренные в нем процедурные конструкции позволяют вполне успешно решать большинство практических задач.

В настоящей главе изложены рекомендации, позволяющие разрабатывать на языке T-SQL максимально эффективные хранимые процедуры, а также приведены основные сведения о том, как для этой цели могут использоваться программные средства инфраструктуры .NET, в частности сборки (assembly). Кроме того, в этой главе описаны формальные параметры, возвращаемые значения, средства управления процессом выполнения, циклические структуры, основные и дополнительные средства перехвата ошибок и т.д. Короче говоря, это — большая глава, в которой рассматриваются многочисленные темы. Но прежде всего в ней приведены основные сведения о создании хранимых процедур.

Основной синтаксис операторов создания хранимых процедур

Операторы создания хранимых процедур во многом напоминают операторы создания любых других объектов базы данных, если не считать того, что в них обязательно применяется ключевое слово AS, которое было впервые упомянуто в данной книге при описании представлений. Основной синтаксис операторов создания хранимых процедур выглядит следующим образом:

```
CREATE PROCEDURE|PROC <sproc name>
   [<parameter name> [schema.]<data type> [VARYING] [= <default value>]
         [OUT [PUT]][,
    <parameter name> [schema.]<data type> [VARYING] [= <default value>]
         [OUT[PUT]][,
    ...
    ...
      ]]
[WITH
   RECOMPILE| ENCRYPTION | [EXECUTE AS { CALLER|SELF|OWNER|<'user name'>}]
[FOR REPLICATION]
AS
   <code> | EXTERNAL NAME <assembly name>.<assembly class>
```

Вполне очевидно, что структура этого оператора также соответствует основному синтаксису CREATE <Object Type> <Object Name>, лежащему в основе любого оператора CREATE. Единственная отличительная особенность этого оператора состоит в том, что в нем допускается использовать ключевое слово PROCEDURE или PROC. Оба эти варианта являются вполне допустимыми, но, как всегда, автор рекомендует придерживаться единообразия в вашем выборе (сам автор предпочитает использовать ключевое слово PROC и немного сокращать объем исходного кода). Имя хранимой процедуры должно соответствовать правилам именования, описанным в главе 1.

Вслед за именем процедуры должен быть приведен список параметров. Применение параметров не является обязательным, но обсуждение этой темы будет продолжено в одном из следующих разделов настоящей главы.

В конце определения хранимой процедуры вслед за ключевым словом AS должен быть приведен действительный код процедуры.

Пример несложной хранимой процедуры

По-видимому, основной синтаксис хранимой процедуры проще всего описать на примере хранимой процедуры самого несложного типа, которая возвращает значения всех полей всех строк таблицы, иными словами, все данные таблицы.

Безусловно, читатель уже не раз использовал в своей практике запросы, которые возвращают все содержимое таблицы (представленные в форме SELECT * FROM...). Такие запросы были впервые описаны в главе, посвященной синтаксису основных запросов. Чтобы создать хранимую процедуру для выполнения столь простого запроса, достаточно ввести текст запроса в той части определения хранимой процедуры, которая предназначена для включения кода:

```
USE Northwind
GO
CREATE PROC spShippers
AS
   SELECT * FROM Shippers
```

В этом сценарии заслуживает внимания то, что оператору CREATE предшествует ключевое слово GO (если бы в состав сценария входил только оператор SELECT, это ключевое слово могло бы не потребоваться). Дело в том, что большинство операторов CREATE, предназначенных для создания объектов, отличных от таблицы, не допускается включать в один пакет с каким-либо другим кодом. В действительности может оказаться рискованной даже попытка применить сценарий создания таблицы, в котором оператор CREATE TABLE не выделен с помощью оператора GO в отдельный пакет. А в рассматриваемом случае наличие оператора USE в одном пакете с оператором CREATE PROC вообще не допускается, поэтому попытка исключить оператор GO привела бы к ошибке.

После создания хранимой процедуры вызовем ее на выполнение, чтобы ознакомиться с полученными результатами:

```
EXEC spShippers
```

Полученные результаты полностью совпадают с теми, которые были бы получены при выполнении отдельно взятого оператора SELECT, который входит в состав хранимой процедуры:

```
ShipperID        CompanyName            Phone
-----------      ----------------       --------------------
1                Speedy Express         (503) 555-9831
2                United Package         (503) 555-3199
3                Federal Shipping       (503) 555-9931
(3 row(s) affected)
```

Итак, первая попытка создать хранимую процедуру оказалась успешной. Это произошло не случайно, и, откровенно говоря, в большинстве ситуаций задача создания хранимых процедур не является такой уж сложной, как хотят внушить непосвященным большинство специалистов по базам данных (чтобы никто не смел подвергнуть сомнению их привилегированное положение). Тем не менее возможности хранимых процедур гораздо шире, чем показано в данном примере, и мы находимся лишь в самом начале пути.

Модификация хранимых процедур с помощью оператора ALTER

Автор должен признаться, что занимался вырезкой и вставкой почти всего текста, приведенного в этом и в следующем ("Удаление хранимых процедур") разделах, взяв данный текст из главы, посвященной представлениям. Я хочу этим сказать, что операторы ALTER и DROP действуют применительно к хранимым процедурам идентично тому, как они применяются к представлениям.

Редактируя хранимые процедуры с помощью операторов T-SQL, следует помнить главное — отредактированная хранимая процедура полностью заменяет существую-

щую. Различия, возникающие при использовании операторов ALTER PROC и операторов CREATE PROC, заключаются лишь в следующем.

- Оператор ALTER PROC рассчитан на то, что хранимая процедура, к которой он применяется, уже существует, а оператор CREATE PROC предназначен для создания хранимой процедуры, которая в текущий момент не существует в базе данных.
- После выполнения оператора ALTER PROC сохраняются все права доступа к рассматриваемой хранимой процедуре, предоставленные пользователям. Этот оператор сохраняет тот же идентификатор объекта, который был присвоен ранее хранимой процедуре, входящей в число других системных объектов, и обеспечивает сохранение зависимостей. Например, если в процедуре A предусмотрен вызов процедуры B, но происходит удаление и повторное создание процедуры B, то зависимость между этими двумя процедурами больше не обнаруживается. Если же для модификации процедуры B используется оператор ALTER PROC, указанная зависимость сохраняется.
- После выполнения оператора ALTER PROC сохраняется вся информация о зависимостях, относящаяся к другим объектам, в которых могут быть предусмотрены вызовы модифицируемой хранимой процедуры.

Второе из этих трех различий является наиболее важным.

> Итак, если для модификации хранимой процедуры выполняется оператор DROP, а затем применяется оператор CREATE, то достигается почти такой же эффект, как при использовании оператора ALTER PROC, если не считать очень важного различия, — после удаления и повторного создания хранимой процедуры необходимо полностью определять заново все права, касающиеся того, кто может и не может применять эту хранимую процедуру.

Удаление хранимых процедур

Оператор удаления хранимой процедуры является очень простым:

```
DROP PROC|PROCEDURE <sproc name>
```

После его выполнения хранимая процедура исчезает из базы данных.

Применение параметров

Хранимая процедура предоставляет определенные процедурные возможности (а если она применяется в инфрастуктуре .NET, такие возможности становятся весьма значительными), а также обеспечивает повышение производительности (дополнительная информация на эту тему приведена ниже), но в большинстве обстоятельств хранимая процедура не позволяет добиться многого, если не предусмотрена возможность передать ей некоторые данные, указывающие на то, какие действия должны быть выполнены с ее помощью. Например, от хранимой процедуры anspDeleteShipper, предназначенной для удаления данных о поставщике, не будет большой пользы, если не предусмотрена возможность передавать ей информацию о том, какая строка со сведениями о поставщике должна быть удалена, поэтому необхо-

димо предусмотреть для хранимой процедуры соответствующие **входные параметры**. Аналогичным образом, часто требуется получить из хранимой процедуры определенную информацию, причем не просто один или несколько наборов строк с данными из таблицы, а более конкретную информацию. Еще одним примером той задачи, которую можно было бы решить с использованием хранимой процедуры, может стать обновление нескольких строк в таблице и получение сведений, ограничивающихся лишь тем, какие именно строки были обновлены. Чаще всего такие данные нелегко получить в форме набора строк, поэтому возникает необходимость предусмотреть использование **выходных параметров**.

При вызове хранимой процедуры параметры могут быть заданы либо с учетом позиции, либо по имени, а в самой вызываемой хранимой процедуре способ, применяемый для передачи параметров, не играет особой роли, поскольку для всех параметров, независимо от способа их передачи в процедуру, используется одинаковый формат объявления.

Объявление параметров

Для объявления параметра необходимо задать от двух до четырех указанных ниже фрагментов информации.

- Имя.
- Тип данных.
- Заданное по умолчанию значение.
- Обозначение выходного параметра.

Объявление параметра имеет следующий синтаксис:

```
@parameter_name [AS] datatype [= default|NULL] [VARYING] [OUTPUT|OUT]
```

Правила именования, в соответствии с которыми формируется имя параметра, являются довольно простыми. В основном они совпадают с правилами составления имен объектов, описанными в главе 1, за исключением того, что в них не допускается использование пробелов. Кроме того, имена параметров должны начинаться со знака `@`.

Тип данных, во многом аналогично имени, должен быть объявлен так же, как и для переменной, – с указанием допустимого встроенного или определяемого пользователем типа данных СУБД SQL Server.

Одно особое требование, которое следует учитывать при объявлении типа данных, состоит в том, что при объявлении параметра типа `CURSOR` необходимо также использовать опции `VARYING` и `OUTPUT`. Способ применения параметров такого типа является довольно необычным, и его описание полностью выходит за рамки настоящей книги, но о нем следует помнить на тот случай, если эта тема будет затронута в документации `Books Online` или в какой-либо другой, чтобы можно было определить общее назначение указанных опций.

Кроме того, следует учитывать, что ключевое слово `OUTPUT` может быть сокращенно записано как `OUT`.

Значительные различия между объявлениями параметров хранимых процедур и объявлениями переменных начинают впервые обнаруживаться, когда дело касается значений, заданных по умолчанию. Прежде всего, при инициализации переменным всегда присваиваются NULL-значения, а на параметры это правило не распространя-

ется. В действительности, если в объявлении параметра не предусмотрено заданное по умолчанию значение, то подразумевается, что этот параметр должен быть обязательным и что при вызове хранимой процедуры должно быть указано его начальное значение. Чтобы задать предусмотренное по умолчанию значение, необходимо добавить знак равенства (=) после обозначения типа данных, а затем указать применяемое по умолчанию значение. Благодаря этому пользователи получают возможность при вызове хранимой процедуры принимать решение о том, следует ли задать другое значение параметра или воспользоваться значением, предусмотренным по умолчанию.

Создадим еще одну хранимую процедуру, но на этот раз предусмотрим использование нескольких входных параметров для ввода новой строки в таблицу Shippers:

```
USE Northwind
GO
CREATE PROC spInsertShipper
  @CompanyName   nvarchar(40),
  @Phone         nvarchar(24)
AS
  INSERT INTO Shippers
  VALUES
    (@CompanyName, @Phone)
```

Предыдущая хранимая процедура позволяла определить, какие данные в настоящее время находятся в таблице Shippers, а новую хранимую процедуру можно использовать для вставки в эту таблицу дополнительных строк, как в следующем примере:

```
EXEC spInsertShipper 'Speedy Shippers, Inc.', '(503) 555-5566'
```

После вызова на выполнение этого оператора в программе Query Analyzer результаты применения хранимой процедуры выглядят так же, как если бы был непосредственно вызван сам оператор INSERT:

```
(1 row(s) affected)
```

Теперь снова вызовем на выполнение первую хранимую процедуру, чтобы ознакомиться с тем, какие изменения произошли в базе данных:

```
EXEC spShippers
ShipperID           CompanyName              Phone
----------------    -------------------      -------------------
1                   Speedy Express           (503) 555-9831
2                   United Package           (503) 555-3199
3                   Federal Shipping         (503) 555-9931
4                   Speedy Shippers, Inc.    (503) 555-5566
(4 row(s) affected)
```

Вполне очевидно, что произошла вставка еще одной строки и в столбец идентификации для этой строки было вставлено новое значение.

В объявлении хранимой процедуры для двух указанных параметров не были предусмотрены значения, применяемые по умолчанию, поэтому оба параметра рассматриваются как обязательные. Это означает, что для успешного вызова хранимой процедуры необходимо предоставить оба параметра. В этом можно легко убедиться, осуществив попытку снова вызвать хранимую процедуру, указав только один параметр, как в следующем примере, или вообще не указывая параметры:

```
EXEC spInsertShipper 'Speedy Shippers, Inc.'
```

СУБД SQL Server немедленно сообщает о том, что возникла ошибка:

```
Msg 201, Level 16, State 4, Procedure spInsertShipper, Line 0
Procedure or Function 'spInsertShipper' expects parameter '@Phone', which
was not supplied.
```

Предоставление значений, применяемых по умолчанию

Чтобы указать, что параметр является необязательным, необходимо ввести для него значение, предусмотренное по умолчанию. Для этого достаточно ввести знак = и задать значение, которое должно применяться по умолчанию, после указания типа данных, но перед запятой.

Подготовим еще один вариант хранимой процедуры, предназначенной для вставки данных о поставщиках, но на этот раз обозначим параметр, соответствующий номеру телефона, как необязательный:

```
USE Northwind
GO

CREATE PROC spInsertShipperOptionalPhone
   @CompanyName   nvarchar(40),
   @Phone         nvarchar(24) = NULL
AS
   INSERT INTO Shippers
   VALUES
      (@CompanyName, @Phone)
```

После этого снова введем тот же оператор, но на этот раз с указанием имени хранимой процедуры:

```
EXEC spInsertShipperOptionalPhone 'Speedy Shippers, Inc'
```

Эта попытка ввода данных осуществляется успешно, и в базу данных вставляется новая строка:

```
(1 row(s) affected)
EXEC spShippers
ShipperID         CompanyName                Phone
------------      ----------------           --------------------
1                 Speedy Express             (503) 555-9831
2                 United Package             (503) 555-3199
3                 Federal Shipping           (503) 555-9931
4                 Speedy Shippers, Inc.      (503) 555-5566
5                 Speedy Shippers, Inc.      NULL
(5 row(s) affected)
```

В данном случае в качестве заданного по умолчанию применяется NULL-значение, но как применяемое по умолчанию может быть задано любое другое значение, совместимое с типом данных параметра, для которого определяется предусмотренное по умолчанию значение. Кроме того, следует отметить, что мы не обязаны задавать предусмотренное по умолчанию значение для обоих параметров, поскольку мы сами вправе решать, для какого из этих параметров должно быть предусмотрено применяемое по умолчанию значение и для какого — нет, т.е. определять, какие параметры являются обязательными (не имеющими заданных по умолчанию значений), а какие — нет (имеющими заданные по умолчанию значения).

Использование выходных параметров

Иногда возникает необходимость передать в тот объект, из которого вызвана хранимая процедура, не набор строк, а какую-то другую информацию. В качестве примера решения такой задачи создадим модифицированную версию на основе двух последних рассматривавшихся в этой главе хранимых процедур.

Предположим, например, что выполняется вставка данных в таблицу (по аналогии с тем, как было сделано в предыдущем примере), но планируется выполнение дополнительной работы с использованием вставленной строки.

Более конкретно укажем, что при каждой вставке новой строки в таблицу `Orders` базы данных `Northwind` необходимо также вставлять строки с данными расшифровки заказов в таблицу `Order Details`. Таблицы `Orders` и `Order Details` связаны между собой, и чтобы не нарушить эту связь, необходимо определить значение в столбце идентификации той строки, которая была только что вставлена в таблицу `Orders`, а затем применить это значение для вставки данных в таблицу `Order Details`. Предназначенная для этого хранимая процедура может выглядеть точно так же, как приведенная выше хранимая процедура `spInsertShipper`, за исключением того, что в ней предусмотрены параметры, соответствующие различным столбцам таблицы и, что важнее всего, эта процедура должна иметь выходной параметр, с помощью которого осуществляется возврат идентификационного значения, полученного при выполнении первой операции вставки:

```
USE Northwind
GO
CREATE PROC spInsertOrder
   @CustomerID       nvarchar(5),
   @EmployeeID       int,
   @OrderDate        datetime       = NULL,
   @RequiredDate     datetime       = NULL,
   @ShippedDate      datetime       = NULL,
   @ShipVia          int,
   @Freight          money,
   @ShipName         nvarchar(40)   = NULL,
   @ShipAddress      nvarchar(60)   = NULL,
   @ShipCity         nvarchar(15)   = NULL,
   @ShipRegion       nvarchar(15)   = NULL,
   @ShipPostalCode   nvarchar(10)   = NULL,
   @ShipCountry      nvarchar(15)   = NULL,
   @OrderID          int            OUTPUT
AS
   /* Создание новой строки */
   INSERT INTO Orders
   VALUES
      (
         @CustomerID,
         @EmployeeID,
         @OrderDate,
         @RequiredDate,
         @ShippedDate,
         @ShipVia,
         @Freight,
         @ShipName,
         @ShipAddress,
```

```
        @ShipCity,
        @ShipRegion,
        @ShipPostalCode,
        @ShipCountry
    )
/* Значение идентификации передается из вновь созданной записи
** в выходную переменную */
SELECT @OrderID = @@IDENTITY
```

Теперь проведем проверку, чтобы узнать, как действует эта процедура, причем лишь на этот раз зададим значения параметров, определяя их по именам, а не по позициям. Для того чтобы определить, как действует выходной параметр, необходимо также подготовить небольшой фрагмент испытательного кода в виде сценария, вызывающего на выполнение хранимую процедуру:

```
USE Northwind
GO
DECLARE  @MyIdent  int
EXEC spInsertOrder
    @CustomerID = 'ALFKI',
    @EmployeeID = 5,
    @OrderDate = '5/1/1999',
    @ShipVia = 3,
    @Freight = 5.00,
    @OrderID = @MyIdent OUTPUT
SELECT @MyIdent AS IdentityValue
SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

Обратите внимание на то, что заданы не все параметры. Некоторые из параметров являются необязательными, поэтому было решено не предоставлять для них значения, в результате чего используются значения, заданные по умолчанию. А если бы вызов хранимой процедуры и передача значений происходили с использованием позиционных параметров, то пришлось бы заполнять каждую позицию в списке параметров по меньшей мере до того, как встретился бы последний параметр, для которого должно быть предусмотрено значение.

Теперь ознакомимся с тем, какие при этом достигаются результаты, но следует учитывать, что идентификационное значение, полученное читателем, может отличаться от полученного автором, в зависимости от того, какие модификации уже были внесены в таблицу `Orders`:

```
(1 row(s) affected)
IdentityValue
-----------------
11078
(1 row(s) affected)
OrderID     CustomerID     EmployeeID     OrderDate                ShipName
---------   -----------    -----------    -------------            --------
11078       ALFKI          5              1999-05-01 00:00:00.000  NULL
(1 row(s) affected)
```

Первая строка с сообщением `row(s) affected` фактически представляет собой реакцию на выполнение самой хранимой процедуры, в результате чего произошла

вставка одной строки. С дугой стороны, второй результирующий набор предоставляет информацию об идентификационном значении, которое использовалось при вставке строки (в примере, выполненном на компьютере автора, это значение равно 11078). Это может служить неопровержимым доказательством того, что идентификационное значение было действительно успешно передано из хранимой процедуры с помощью выходного параметра. Наконец, была выполнена выборка данных нескольких полей из строки таблицы Orders для проверки того, что при вставке этой строки действительно использовались такие данные, как и следовало ожидать.

Ниже описаны дополнительные требования, которые распространяются и на саму хранимую процедуру, и на то, как она должна использоваться в вызывающем сценарии.

- Ключевое слово OUTPUT для выходного параметра в объявлении хранимой процедуры является обязательным.
- При вызове хранимой процедуры ключевое слово OUTPUT должно использоваться так же, как и при объявлении этой хранимой процедуры. Благодаря этому СУБД SQL Server получает заблаговременное предупреждение о том, что соответствующий параметр потребует специальной обработки. Но следует учитывать, что упущение, из-за которого во время вызова хранимой процедуры выходной параметр не будет обозначен ключевым словом OUTPUT, не приведет к возникновению ошибки этапа прогона (не появятся какие-либо сообщения о такой ошибке), но значение выходного параметра не будет передано в предоставленную вами переменную (в конечном итоге значение переменной останется тем же, каким оно уже является, т.е. равным NULL). Это означает, что указанное упущение приводит к тому нарушению в работе программного обеспечения, которое автор считает наиболее опасным, – к получению непредсказуемых результатов.
- Переменная, которой присваивается результат вычисления значения выходного параметра, не обязательно должна иметь то же имя, что и внутренний параметр хранимой процедуры. Например, в приведенной выше хранимой процедуре внутренний параметр имел имя @OrderID, а переменная, которой присваивалось значение этого параметра, имела имя @MyIdent.
- В рассматриваемом сценарии ключевое слово EXEC (или EXECUTE) было обязательным, поскольку вызов хранимой процедуры не представлял собой первый оператор в пакете (в вызове хранимой процедуры ключевое слово EXEC можно не задавать, если этот вызов является первым оператором в пакете), но сам автор рекомендует неизменно применять это ключевое слово, невзирая на то, является ли оно обязательным или не обязательным.

Операторы управления ходом выполнения

Операторы управления ходом выполнения являются неотъемлемым атрибутом любого современного процедурного языка программирования. Трудно себе представить, как можно было бы без помощи таких операторов управлять последовательностью выполнения различных фрагментов кода с учетом условий. В языке T-SQL предусмотрена бо́льшая часть программных средств управления ходом выполнения программы, включая перечисленные ниже.

- Оператор IF...ELSE.
- Оператор GOTO.
- Оператор WHILE.
- Оператор WAITFOR.
- Оператор TRY/CATCH.

Кроме того, в языке T-SQL предусмотрен оператор CASE (аналогичный операторам SELECT CASE, DO CASE и SWITCH/BREAK в других языках), но он не обеспечивает такого же уровня управления программой, как операторы других языков.

Оператор IF...ELSE

Операторы IF...ELSE действуют в языке T-SQL в основном так же, как и в любых других языках, но, по мнению автора, способ их реализации в большей степени напоминает реализацию в языке C. Эти операторы имеют следующий основной синтаксис:

```
IF <Boolean Expression>
  <SQL statement> | BEGIN <code series> END
[ELSE
  <SQL statement> | BEGIN <code series> END]
```

В качестве выражения <Boolean Expression> может быть задано практически любое выражение, результат вычисления которого приводит к возврату булева значения.

При использовании оператора IF...ELSE также необходимо избегать одной из ловушек, в которые, по мнению автора, наиболее часто попадают программисты, работающие на языке SQL, — неправильное использование NULL-значений. Трудно представить себе, насколько часто приходится заниматься отладкой хранимых процедур, в результате которой обнаруживается такой оператор:

```
IF @myvar = NULL
```

Безусловно, выражение @myvar = NULL никогда не может принять истинное значение в большинстве систем (как описано ниже), и в конечном итоге пропускаются все значения переменной @myvar, равные NULL. Вместо этого в качестве указанного оператора следует использовать такую конструкцию:

```
IF @myvar IS NULL
```

Не забывайте, что NULL-значение ничему не равно, даже другому NULL-значению, поэтому вместо операции = необходимо использовать операцию IS.

Необходимо отметить, что сказанное выше может оказаться неверным, если опции конфигурации ANSI_NULLS присвоено значение OFF. По умолчанию эта функция имеет значение ON, и в таком случае проверка наличия NULL-значения с помощью оператора =, как уже было сказано, не допускается. Но возможность использовать для этого оператор = можно получить, задав значение опции ANSI_NULLS, равное OFF. Тем не менее автор настоятельно рекомендует не делать этого, поскольку такое действие равносильно нарушению стандарта ANSI (а также просто приводит к получению неправильных результатов).

Следует учитывать, что выполняемым по условию считается только тот оператор, который непосредственно следует за оператором IF (ближайшим к нему). Вместо одного оператора можно предусмотреть выполнение по условию нескольких опера-

торов, включив их в состав блока управления процессом выполнения с помощью конструкции BEGIN...END (дополнительная информация на эту тему приведена ниже в данной главе).

Создадим новый вариант запроса, который рассматривался в предыдущем примере, с учетом той ситуации, что программа позволяет вводить заказы со значением даты OrderDate, относящимся к тем прошедшим суткам, которые выходят за пределы допустимого периода.

Допустим, что к вам обратился коммерческий директор и описал недопустимую ситуацию, возникающую в системе в связи с тем, что какие-то недобросовестные сотрудники вводят заказы, относящиеся к прошедшей дате, далеко выходящей за пределы того периода, за который коммерческий директор проводит анализ сбыта по данным из заказов. Поэтому коммерческий директор установил новое правило, согласно которому заказ должен быть введен в систему в течение семи суток со времени его приема; в противном случае дата заказа рассматривается как недействительная и вместо нее в базу данных должно быть записано NULL-значение.

Чтобы вместо недопустимой даты заказа записать NULL-значение, можно воспользоваться оператором IF...ELSE.

Для этого необходимо провести простую проверку, для чего можно применить функцию DATEDIFF, которая имеет следующий синтаксис:

```
DATEDIFF(<datepart>, <startdate>, <enddate>)
```

Функция DATEDIFF сравнивает две даты. В рассматриваемом случае для сравнения используются введенная сотрудником дата заказа и текущая дата. Но фактически с помощью функции DATEDIFF можно провести сравнение любых компонентов данных <datepart> в формате datetime, заданных в качестве параметров, начиная с годов и заканчивая миллисекундами. В данном случае достаточно указать в качестве обозначения компонента даты константу dd, обозначающую сутки, и включить полученное выражение в оператор IF:

```
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
```

Если возвращаемое значение функции DATEDIFF больше 7 (иначе говоря, если заказ введен больше 7 дней тому назад), то вместо заданного в нем значения даты заказа необходимо ввести NULL:

```
SELECT @OrderDate = NULL
```

После того как оператор IF успешно подготовлен, можно приступить к написанию новой версии хранимой процедуры spInsertOrder:

```
USE Northwind
GO
CREATE PROC spInsertDateValidatedOrder
  @CustomerID       nvarchar(5),
  @EmployeeID       int,
  @OrderDate        datetime       = NULL,
  @RequiredDate     datetime       = NULL,
  @ShippedDate      datetime       = NULL,
  @ShipVia          int,
  @Freight          money,
  @ShipName         nvarchar(40)   = NULL,
  @ShipAddress      nvarchar(60)   = NULL,
  @ShipCity         nvarchar(15)   = NULL,
```

```
    @ShipRegion       nvarchar(15)   = NULL,
    @ShipPostalCode   nvarchar(10)   = NULL,
    @ShipCountry      nvarchar(15)   = NULL,
    @OrderID          int            OUTPUT
AS
/* Проверка того, не отстоит ли дата поставки больше чем на семь дней
** от текущей даты и в случае положительного ответа ее замена NULL-значением
*/
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
   SELECT @OrderDate = NULL
/* Создание новой записи */
INSERT INTO Orders
VALUES
(
   @CustomerID,
   @EmployeeID,
   @OrderDate,
   @RequiredDate,
   @ShippedDate,
   @ShipVia,
   @Freight,
   @ShipName,
   @ShipAddress,
   @ShipCity,
   @ShipRegion,
   @ShipPostalCode,
   @ShipCountry
)
/* Значение идентификации передается из вновь созданной записи
** в выходную переменную */
SELECT @OrderID = @@IDENTITY
```

Теперь попытаемся вызвать на выполнение тот же сценарий проверки, который применялся вместе с первоначальным вариантом хранимой процедуры `spInsertOrder`, но с небольшими изменениями, позволяющими учесть новые требования:

```
USE Northwind
GO
DECLARE   @MyIdent   int

EXEC spInsertDateValidatedOrder
   @CustomerID = 'ALFKI',
   @EmployeeID = 5,
   @OrderDate = '5/1/1999',
   @ShipVia = 3,
   @Freight = 5.00,
   @OrderID = @MyIdent OUTPUT
SELECT @MyIdent AS IdentityValue
SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

На этот раз данные, вводимые в базу данных, изменяются даже несмотря на то, что основная часть хранимой процедуры осталась прежней, поэтому результаты выборки содержат следующее:

```
(1 row(s) affected)
IdentityValue
-----------------
11079
(1 row(s) affected)
OrderID     CustomerID     EmployeeID     OrderDate          ShipName
---------   -----------    -----------    -------------      -----------
11079       ALFKI          5              NULL               NULL
(1 row(s) affected)
```

В этот раз была задана такая же дата, как и в предыдущем примере (`5/1/1999`), но в базу данных вставлено другое значение, поскольку с помощью оператора `IF` недопустимое значение было изъято и заменено перед вставкой данных в базу данных.

Конструкция ELSE

Описанная выше на примере замены даты возможность исправлять вводимые данные действительно является очень удобной, но не подходит для всех сценариев, которые могут применяться на практике. При проверке условий оператора `IF` очень часто (а фактически чаще всего) возникает необходимость выполнить определенный ряд операторов, если результат проверки окажется истинным, а также отдельный ряд операторов, если результат будет ложным. В последнем случае может применяться конструкция `ELSE`.

Иногда возникают ситуации, в которых результаты проверки не могут быть выражены в виде булева значения, т.е. результат неизвестен (например, если применяется сравнение со NULL-значением). Любые выражения, которые возвращают результат, рассматриваемый как неопределенный, считаются возвращающими значение FALSE.

Конструкция `ELSE` оператора `IF` действует во многом аналогично тому, как и в других языках. При этом независимо от небольших различий в синтаксисе основное назначение этой конструкции остается одним и тем же — выполняются либо операторы в конструкции `ELSE`, либо операторы в конструкции `IF`.

Чтобы немного дополнить рассматриваемый пример, займемся поиском строк за самую старую дату, которые в настоящее время находятся в таблице `Orders` базы данных `Northwind`:

```
USE Northwind
GO
SELECT TOP 5 OrderID, OrderDate
FROM Orders
WHERE OrderDate IS NOT NULL
ORDER BY OrderDate
```

Эти результаты содержат кое-что интересное:

```
OrderID          OrderDate
-----------      -------------------------------
10248            1996-07-04 00:00:00.000
10249            1996-07-05 00:00:00.000
10250            1996-07-08 00:00:00.000
10251            1996-07-08 00:00:00.000
10252            1996-07-09 00:00:00.000
(5 row(s) affected)
```

Дело в том, то ни одна из этих дат не имеет компонента с обозначением времени. С формальной точки зрения можно считать, что временем, к которому относятся все заказы, является полночь, но мне кажется, что читатель уже догадался, в чем тут дело. Скорее всего, из значений даты заказов компонент с указанием времени был удален преднамеренно, поскольку сравнивать только даты (не содержащие сведений о времени) гораздо проще.

Поэтому перед нами стоит задача доработать рассматриваемую хранимую процедуру таким образом, чтобы с ее помощью обеспечивалось сохранение всех временных отметок исключительно в виде дат, т.е. без указания времени. Текущий вариант хранимой процедуры не соответствует этому требованию, поскольку в нем предусмотрена вставка всего значения даты, включая время, но проверим, так ли это дело обстоит на самом деле:

```
USE Northwind
GO
DECLARE     @MyIdent    int
DECLARE     @MyDate     smalldatetime
SELECT @MyDate = GETDATE()

EXEC spInsertDateValidatedOrder
   @CustomerID = 'ALFKI',
   @EmployeeID = 5,
   @OrderDate  = @MyDate,
   @ShipVia    = 3,
   @Freight    = 5.00,
   @OrderID    = @MyIdent OUTPUT
SELECT @MyIdent AS IdentityValue
SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

Действительно, после вставки даты в этом значении содержится также информация о времени:

```
(1 row(s) affected)
IdentityValue
-------------
11080
(1 row(s) affected)
OrderID     CustomerID     EmployeeID     OrderDate                ShipName
---------   ------------   -----------    -----------              --------
11080       ALFKI          5              2000-07-22 16:48:00.000  NULL
(1 row(s) affected)
```

Таким образом, в разрабатываемой процедуре необходимо учесть два условия. В соответствии с первым условием вместо значения даты необходимо ввести NULL, а в соответствии со вторым — отсечь в значении даты компонент с обозначением времени. К сожалению, в программном обеспечении СУБД SQL Server не предусмотрена функция, позволяющая осуществлять это действие автоматически (по моему скромному мнению — это серьезное упущение). Тем не менее, как описано в следующем разделе, из такого положения можно найти выход.

Усечение компонента поля с типом данных datetime, в котором представлено обозначение времени

Для того чтобы удалить ненужный компонент в значении даты, можно либо разобрать значение даты на компоненты, а затем снова собрать без включения компонента с данными о времени, либо (как предпочитает автор) применить для этой цели функцию CONVERT, чтобы преобразовать дату из формата datetime в строковый формат, не содержащий обозначения времени, после чего выполнить обратное преобразование.

Функция CONVERT() относится к числу удобных многочисленных функций, которые входят в состав программного обеспечения СУБД SQL Server. Первоначально единственный способ преобразования дат из одного типа данных в другой состоял в использовании этой функции. Тем не менее в наши дни функция CONVERT должна применяться в сценариях все реже и реже, поскольку основная часть ее функциональных возможностей продублирована в функции CAST(), которая является совместимой со стандартом ANSI (тогда как функция CONVERT не является таковой). Несмотря на вышесказанное, функция CONVERT предоставляет некоторые специальные возможности форматирования даты, которые не могут быть реализованы с помощью функции CAST.

Функция CONVERT имеет следующий синтаксис:

```
CONVERT(<target data type>, <expression to be converted>, <style>)
```

Первые два параметра почти не нуждаются в пояснении, а последний параметр, <style>, следует описать более подробно. Прежде всего, этот параметр применяется только для преобразования дат. Кроме того, он предназначен для передачи СУБД SQL Server указания на то, в каком формате представлена дата. В качестве примеров широко применяемых форматов даты можно указать 1 (стандартный формат mm/dd/yy, применяемый в США) и 12 (формат yymmdd, определяемый стандартом ISO). После добавления числа 100 к обозначению любого из этих форматов формируется обозначение, которое предусматривает указание полного столетия в формате даты (например, параметр style, равный 101, соответствует применяемому в США стандартному формату с четырехзначным обозначением года, mm/dd/yyyy).

Например, для преобразования даты, полученной с помощью функции GETDATE, может использоваться примерно такая конструкция:

```
SELECT CONVERT(datetime,(CONVERT(varchar,GETDATE(),112)))
```

С помощью этой конструкции осуществляется прямое преобразование в формат даты ANSI, а затем происходит обратное преобразование, в результате чего значение времени отсекается:

```
-----------------------------------
2000-06-06 00:00:00.000
(1 row(s) affected)
```

Вариант разрабатываемой хранимой процедуры с конструкцией ELSE

В предыдущих разделах рассматривались отдельные доработки, которые должны быть внесены в разрабатываемую хранимую процедуру, а в настоящем разделе показано, как объединить их в виде окончательного варианта хранимой процедуры, который соответствует предъявленным требованиям. На этот раз вместо создания отдельной процедуры будет модифицирована существующая хранимая процедура с помощью оператора ALTER. Но следует учитывать, что даже при использовании оператора ALTER необходимо полностью переопределять модифицируемую процедуру:

```
USE Northwind
GO
ALTER PROC spInsertDateValidatedOrder
  @CustomerID       nvarchar(5),
  @EmployeeID       int,
  @OrderDate        datetime       = NULL,
  @RequiredDate     datetime       = NULL,
  @ShippedDate      datetime       = NULL,
  @ShipVia          int,
  @Freight          money,
  @ShipName         nvarchar(40)   = NULL,
  @ShipAddress      nvarchar(60)   = NULL,
  @ShipCity         nvarchar(15)   = NULL,
  @ShipRegion       nvarchar(15)   = NULL,
  @ShipPostalCode   nvarchar(10)   = NULL,
  @ShipCountry      nvarchar(15)   = NULL,
  @OrderID          int            OUTPUT
AS
/* Внесение изменений во входные параметры не рекомендуется - практика
** показывает, что отладка упрощается, если в любое время могут быть проверены их
** первоначальные значения. Поэтому объявлена отдельная переменная, которой
** присваивается конечное значение, предназначенное для вставки в таблицу */
DECLARE  @InsertedOrderDate  smalldatetime

/* Проводится проверка для определения того, не отстоит ли дата поставки больше
** чем на семь дней от текущей даты и в случае положительного ответа эта дата
** заменяется NULL-значением. В противном случае устанавливается значение
** времени, соответствующее полуночи */
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
  SELECT @InsertedOrderDate = NULL
ELSE
  SELECT @InsertedOrderDate =
    CONVERT(datetime,(CONVERT(varchar,@OrderDate,112)))

  /* Создание новой записи */
INSERT INTO Orders
VALUES
(
  @CustomerID,
  @EmployeeID,
  @InsertedOrderDate,
  @RequiredDate,
  @ShippedDate,
  @ShipVia,
  @Freight,
  @ShipName,
  @ShipAddress,
  @ShipCity,
  @ShipRegion,
  @ShipPostalCode,
  @ShipCountry
)
/* Значение идентификации передается из вновь созданной записи
** в выходную переменную */
SELECT @OrderID = @@IDENTITY
```

Если после этого будет снова вызван на выполнение пакет, применявшийся для проверки первоначального варианта процедуры, то будет получен требуемый результат:

```
(1 row(s) affected)
IdentityValue
-------------
11081
(1 row(s) affected)
OrderID     CustomerID    EmployeeID  OrderDate                ShipName
--------    ------------  ----------  -----------------        ---------
11081       ALFKI         5           2000-07-22 00:00:00.000  NULL
(1 row(s) affected)
```

Таким образом, создана хранимая процедура, которая обеспечивает вставку различных данных с учетом того, какие конкретные значения были ей предоставлены для ввода в базу данных.

Внимательный читатель должен заметить, что в этой версии хранимой процедуры не только внесены изменения в код оператора IF. . . ELSE, но и для хранения даты заказа объявлена отдельная переменная.

Такая доработка внесена с учетом того общего подхода в отношении изменения значений входных параметров, которого придерживается автор. По моему мнению, в хранимой процедуре не следует изменять непосредственно сами значения параметров, за исключением тех случаев, когда сразу же после изменения значения входного параметра осуществляется его возврат. Прежде всего, такой подход позволяет создавать более понятный код; по возможности сосредоточив все операторы объявления переменных и присваивания им значения в одном месте, можно избежать необходимости выполнять поиск этих важных операторов по всему коду. Еще одна причина, по-видимому, является гораздо более убедительной — упрощение отладки. Я при любой возможности стараюсь сохранить неизменными входные значения хранимой процедуры, чтобы при проведении отладки можно было легко сравнить входные значения с теми данными в различных местах кода, в сочетании с которыми используются входные значения. Иными словами, я стремлюсь упростить проверку того, правильно ли функционирует хранимая процедура, и чем обусловлено нарушение в ее работе.

Группирование операторов в блоки

Иногда возникает необходимость объединить группу операторов в блок, чтобы можно было рассматривать их как один оператор (и выполнять по условию либо все операторы блока, либо не выполнять ни одного из них). Например, синтаксическая конструкция оператора IF предусматривает выполнение по условию только одного оператора, а именно, того, который следует за ним. Если бы не было возможности выполнить по условию сразу несколько операторов, то пришлось бы задавать отдельный оператор IF с одним и тем же условием для каждого оператора, который должен быть выполнен, если результат проверки условия окажется истинным.

К счастью, в языке SQL предусмотрен способ группирования операторов в виде блоков. Каждый блок, созданный таким образом, рассматривается как один оператор. Определение блока начинается с конструкции BEGIN и включает в себя все следующие операторы вплоть до конструкции END. Ниже приведен пример сложного выражения, в котором используются блоки, состоящего из нескольких операторов IF.

```
IF <Expression1>
BEGIN   -- Начинается первый блок кода, который выполняется, только если
```

```
      -- выражение Expression1 имеет значение TRUE
   Операторы, которые выполняются, если выражение Expression1 имеет значение TRUE
   IF <Expression2>   -- Выполняется, только если этот блок активен
     BEGIN
       Операторы, которые выполняются, если выражения Expression1
       и Expression2 имеют значение TRUE
     END
   Операторы, которые также относятся к первому блоку
END   -- Эта конструкция обозначает конец внешнего блока
ELSE
BEGIN
   Операторы, которые выполняются, если выражение Expression1 имеет значение FALSE
END
```

Обратите внимание на то, что благодаря применению блоков появляется возможность создавать вложенные программные конструкции. Каждый внутренний блок должен быть полностью заключен во внешний блок. Автору еще не приходилось слышать о том, что есть какой-то предел глубины, на которую допускается вкладывать блоки BEGIN...END, но обязан порекомендовать, чтобы такое значение глубины вложенности не выходило за разумные рамки. Дело в том, что на практике код, в котором применяется слишком большая глубина вложенности, становится сложным для восприятия, причем исправить эту ситуацию не позволяет даже очень тщательное форматирование кода.

Исключительно ради проверки возможности использования блоков внесем еще одно изменение в последний вариант хранимой процедуры, используемой для вставки данных о заказах. Но на этот раз в ходе доработки кода предусмотрена возможность предоставления пользователю небольшого дополнительного объема полезной информации в комментариях по поводу того, как происходит изменение входного параметра хранимой процедуры. Созданная таким образом процедура может использоваться как своего рода вступление для приведенного ниже раздела, посвященного обработке ошибок.

Итак, каждый раз, когда в хранимой процедуре будет принято решение перед вставкой в базу данных заменить дату, введенную пользователем, каким-то другим значением, необходимо также сообщить пользователю, какие именно действия по замене даты будут выполнены. Для вывода информации о конкретных выполняемых действиях используется оператор PRINT. Вывод информации осуществляется с учетом текущей ситуации, для чего соответствующие операторы PRINT вводятся в состав кода операторов IF...ELSE. Обратите внимание на то, что операторы PRINT не формируют какие-либо сообщения об ошибках, а просто выводят текстовую информацию, касающуюся сложившейся ситуации возникновения ошибки.

Дополнительная информация на эту тему приведена в разделе, посвященном обработке ошибок.

Рассматриваемый вариант хранимой процедуры приведен ниже.

```
USE Northwind
GO
ALTER PROC spInsertDateValidatedOrder
   @CustomerID             nvarchar(5),
   @EmployeeID             int,
   @OrderDate        datetime       = NULL,
   @RequiredDate     datetime       = NULL,
```

```
    @ShippedDate      datetime       = NULL,
    @ShipVia          int,
    @Freight          money,
    @ShipName         nvarchar(40)   = NULL,
    @ShipAddress      nvarchar(60)   = NULL,
    @ShipCity         nvarchar(15)   = NULL,
    @ShipRegion       nvarchar(15)   = NULL,
    @ShipPostalCode   nvarchar(10)   = NULL,
    @ShipCountry      nvarchar(15)   = NULL,
    @OrderID          int            OUTPUT
AS
/* Внесение изменений во входные параметры не рекомендуется - практика
** показывает, что отладка упрощается, если в любое время могут быть проверены их
** первоначальные значения. Поэтому объявлена отдельная переменная, которой
** присваивается конечное значение, предназначенное для вставки в таблицу */
DECLARE   @InsertedOrderDate   smalldatetime
/* Проводится проверка для определения того, не отстоит ли дата поставки больше
** чем на семь дней от текущей даты и в случае положительного ответа эта дата
** заменяется NULL-значением. В противном случае устанавливается значение
** времени, соответствующее полуночи */
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
BEGIN
   SELECT @InsertedOrderDate = NULL
   PRINT 'Invalid Order Date'
   PRINT 'Supplied Order Date was greater than 7 days old.'
   PRINT 'The value has been reset to NULL'
END
ELSE
BEGIN
   SELECT @InsertedOrderDate =
      CONVERT(datetime,(CONVERT(varchar,@OrderDate,112)))
      PRINT 'The Time of Day in Order Date was truncated'
END

/* Создание новой записи */
INSERT INTO Orders
VALUES
(
   @CustomerID,
   @EmployeeID,
   @InsertedOrderDate,
   @RequiredDate,
   @ShippedDate,
   @ShipVia,
   @Freight,
   @ShipName,
   @ShipAddress,
   @ShipCity,
   @ShipRegion,
   @ShipPostalCode,
   @ShipCountry
)
/* Значение идентификации передается из вновь созданной записи
** в выходную переменную */
SELECT @OrderID = @@IDENTITY
```

После вызова на выполнение того же пакета, который применяется для проверки, снова будет получен немного другой результат. Прежде всего, в проверочном пакете используется текущая дата:

```
USE Northwind
GO
DECLARE     @MyIdent    int
DECLARE     @MyDate     smalldatetime
SELECT @MyDate = GETDATE()
EXEC spInsertDateValidatedOrder
   @CustomerID = 'ALFKI',
   @EmployeeID = 5,
   @OrderDate  = @MyDate,
   @ShipVia    = 3,
   @Freight    = 5.00,
   @OrderID    = @MyIdent OUTPUT
SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

Обратите внимание на то, что в этом проверочном пакете в целях сокращения был удален оператор `SELECT @MyIdent AS IdentityValue`.

Вполне очевидно, что при использовании новой версии хранимой процедуры не только происходит усечение фактически полученного значения даты, но и формируется сообщение, в котором явно указано, какое действие было выполнено:

```
The Time of Day in Order Date was truncated
(1 row(s) affected)
OrderID     CustomerID     EmployeeID  OrderDate                 ShipName
---------   -----------    ----------  ------------------------  ---------
11080       ALFKI          5           1999-08-30 00:00:00.000   NULL
(1 row(s) affected)
```

После этого вызовем на выполнение предпоследнюю версию проверочного пакета, в которой предусмотрен ввод даты заказа за предыдущий день вручную:

```
USE Northwind
GO
DECLARE  @MyIdent        int
EXEC spInsertDateValidatedOrder
   @CustomerID = 'ALFKI',
   @EmployeeID = 5,
   @OrderDate  = '1/1/1999',
   @ShipVia    = 3,
   @Freight    = 5.00,
   @OrderID    = @MyIdent OUTPUT
SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID  = @MyIdent
```

В этом случае поступает также явное указание на то, какие изменения произошли с введенными данными:

```
Invalid Order Date
Supplied Order Date was greater than 7 days old.
```

```
The value has been reset to NULL
(1 row(s) affected)
OrderID     CustomerID     EmployeeID     OrderDate      ShipName
---------   ------------   ------------   -----------    --------------
11085       ALFKI          5              NULL           NULL
(1 row(s) affected)
```

Оператор CASE

Оператор CASE в определенном смысле является эквивалентным нескольким разным операторам в различных процедурных языках программирования. К числу операторов процедурных языков программирования, которые действуют аналогично оператору CASE, относятся следующие:

- Оператор `switch` в языках программирования C, C++, Delphi.
- Оператор `Select Case` в языке программирования Visual Basic.
- Оператор `Do Case` в языке программирования XBase.
- Оператор `Evaluate` в языке программирования COBOL.

Автор знает о том, что есть также многие другие языки программирования с аналогичными конструкциями, а выше приведены примеры лишь из тех языков, с которыми ему довелось работать в течение многих лет. Реализация оператора CASE в языке T-SQL имеет один значительный недостаток, связанный с тем, что этот оператор в большей степени предназначен для подстановки значений, чем для управления процессом выполнения.

Синтаксическая структура оператора CASE имеет две формы – с входным выражением или с булевым выражением. В первой форме используется входное выражение, которое сравнивается со значением, заданным в каждой из нескольких конструкций WHEN. В документации SQL Server оператор в этой форме именуется **простым оператором CASE**:

```
CASE <input expression>
WHEN <when expression> THEN <result expression>
[...n]
[ELSE <result expression>]
END
```

Во втором варианте синтаксической структуры в каждой конструкции WHEN должно быть предусмотрено выражение, вычисление которого приводит к получению значения TRUE или FALSE. В документации оператор в этой форме называется **поисковым оператором CASE**:

```
CASE
WHEN <Boolean expression> THEN <result expression>
[...n]
[ELSE <result expression>]
END
```

По-видимому, наиболее удобной особенностью оператора CASE является то, что он может использоваться как "встроенный" в оператор SELECT (т.е. может составить неотъемлемую часть указанного оператора). Благодаря этому с помощью оператора CASE можно создавать весьма мощные конструкции.

Отложим на время описание последнего варианта синтаксической структуры, поискового оператора CASE (мы еще посвятим ему несколько слов), и перейдем к рассмотрению простого оператора CASE с нескольких различных точек зрения.

Простой оператор CASE

В основе простого оператора CASE лежит входное выражение, <input expression>, вычисление которого приводит к получению однозначного результата. Сразу же перейдем к рассмотрению примера:

```
USE Northwind
GO
SELECT TOP 10 OrderID, OrderID % 10 AS 'Last Digit', Position =
CASE OrderID % 10
   WHEN 1 THEN 'First'
   WHEN 2 THEN 'Second'
   WHEN 3 THEN 'Third'
   WHEN 4 THEN 'Fourth'
   ELSE 'Something Else'
END
FROM Orders
```

Напомним, что операция % представляет собой **операцию деления по модулю**. Операция деления по модулю выполняется аналогично операции деления (/), но возвращает только остаток от деления. Поэтому 16 % 4 = 0 (16 делится на 4 без остатка), а 16 % 5 = 1 (остаток от деления 16 на 5 равен 1). В данном примере применяется деление по модулю на десять, поэтому выполнение данной операции приводит к получению последней цифры числа, используемого в качестве входных данных.

Рассмотрим, какие результаты будут получены при выполнении этого примера кода:

```
OrderID     Last Digit  Position
---------   ----------  --------------------
10249       9           Something Else
10251       1           First
10258       8           Something Else
10260       0           Something Else
10265       5           Something Else
10267       7           Something Else
10269       9           Something Else
10270       0           Something Else
10274       4           Fourth
10275       5           Something Else
(10 row(s) affected)
```

Обратите внимание на то, что при обнаружении любого значения, соответствующего одному из указанных в конструкциях WHEN, выполняется соответствующая конструкция THEN. А поскольку в операторе предусмотрена также конструкция ELSE, то вместо любого значения, не соответствующего одному из указанных в конструкциях WHEN, возвращается значение, предусмотренное в конструкции ELSE. Если бы конструкция ELSE была исключена, то вместо любого такого значения было бы возвращено NULL-значение.

Рассмотрим еще один пример, который показывает, что возможности выбора способа применения конструкции CASE еще шире. На этот раз в запросе рассматриваются данные из другого столбца:

```
USE Northwind
GO
SELECT TOP 10 OrderID % 10 AS "Last Digit",
   ProductID,
   "How Close?" = CASE OrderID % 10
      WHEN ProductID THEN 'Exact Match!'
      WHEN ProductID - 1 THEN 'Within 1'
      WHEN ProductID + 1 THEN 'Within 1'
      ELSE 'More Than One Apart'
   END
FROM [Order Details]
WHERE ProductID < 10
ORDER BY OrderID DESC
```

Обратите внимание на то, что в данном примере выражения используются не только там, где обычно, но и в конструкциях WHEN, и все же созданный в итоге оператор функционирует успешно:

```
Last Digit     ProductID     How Close?
-----------    -----------   ----------------------------
7              8             Within 1
7              7             Exact Match!
7              6             Within 1
7              4             More Than One Apart
7              3             More Than One Apart
7              2             More Than One Apart
6              6             Exact Match!
5              2             More Than One Apart
2              2             Exact Match!
1              7             More Than One Apart
(10 row(s) affected)
```

Таким образом, при условии, что вычисление выражения в конструкции WHEN приводит к получению конкретного значения, имеющего тип, совместимый с типом входного выражения, обеспечивается сопоставление этого значения со значением входного выражения и применяется соответствующая конструкция THEN.

Поисковый оператор CASE

Оператор CASE этого типа действует в основном так же, как простой оператор CASE, если не считать двух небольших описанных ниже отличий.

- В нем отсутствует входное выражение (напомним, что в простом операторе CASE входное выражение находится между ключевым словом CASE и первым оператором WHEN).
- Вычисление выражения в конструкции WHEN должно приводить к получению булева значения (напомним, что в примерах применения простого оператора CASE, приведенных выше, использовались такие значения, как 1, 3 и ProductID + 1).

Но сам автор считает наиболее привлекательной особенностью этого варианта конструкции CASE то, что она позволяет полностью изменить подход к формированию выражений и применять любые сочетания и комбинации выражений с данными из разных столбцов в зависимости от возможных конкретных ситуаций.

Как обычно, рассмотрим пример, который позволяет лучше всего проиллюстрировать особенности работы оператора:

```
USE Northwind
GO
SELECT TOP 10 OrderID % 10 AS "Last Digit",
   ProductID,
   "How Close?" = CASE
      WHEN (OrderID % 10) < 3 THEN 'Ends With Less Than Three'
      WHEN ProductID = 6 THEN 'ProductID is 6'
      WHEN ABS(OrderID % 10 - ProductID) <= 1 THEN 'Within 1'
      ELSE 'More Than One Apart'
   END
FROM [Order Details]
WHERE ProductID < 10
ORDER BY OrderID DESC
```

Этот пример существенно отличается от приведенных выше примеров применения простых операторов CASE, но его выполнение приводит к успешному получению желаемых результатов:

```
Last Digit     ProductID      How Close?
------------   ------------   ----------------------------------
7              8              Within 1
7              7              Within 1
7              6              ProductID is 6
7              4              More Than One Apart
7              3              More Than One Apart
7              2              More Than One Apart
6              6              ProductID is 6
5              2              More Than One Apart
2              2              Ends With Less Than Three
1              7              Ends With Less Than Three
(10 row(s) affected)
```

При изучении того, как именно происходит вычисление выражений в СУБД SQL Server, необходимо обратить особое внимание на описанные ниже нюансы.

- Даже если значение TRUE может быть получено при вычислении выражений в нескольких конструкциях WHEN, учитывается только результат вычисления в первой по порядку конструкции. Например, предпоследняя строка соответствует условию и в первой конструкции WHEN (последняя цифра меньше 3), и в третьей конструкции (последняя цифра отличается от значения ProductID на 1). Таким же образом действуют аналогичные операторы во многих других языках, включая Visual Basic. Но в языке C выполнение подобного оператора организовано иначе, и об этом должны помнить программисты, работающие на языке C; кроме того, не требуется оператор break, поскольку окончание конструкции THEN всегда обозначается началом следующей конструкции.

- В условных выражениях могут использоваться различные комбинации и сочетания значений полей. В данном примере использовались значения `OrderID`, `ProductID`, а также оба значения вместе.
- Возможности использования практически любых выражений буквально неограниченны, при условии, что в конечном итоге вычисление конкретного выражения приведет к получению булева результата.

Рассмотрим описанные выше особенности поискового оператора `CASE` на несколько более сложном примере. В этом примере не рассматриваются комбинации и сочетания значений из разных столбцов; вместо этого используется только один столбец, в котором осуществляется поиск (можно было бы применять для проверки и другие столбцы, но чаще всего необходимость в этом не возникает). Автор продемонстрирует ситуацию из реальной жизни, в которой он помог найти решение для довольно крупного узла электронной коммерции.

Сценарий заключается в следующем: в торговле обычно принято назначать легко запоминающиеся цены, поскольку покупателям не нравится, когда, допустим, из-за 10%-й наценки цена на товары принимает вид `$10.13`, `$23.19` и т.д. Поэтому маркетологи рекомендуют округлять цены так, чтобы они оканчивались цифрами наподобие `49`, `75`, `95` или `99`. В рассматриваемом сценарии было предъявлено требование — подготовить для анализа новый прейскурант с ценами, не выходящими за пределы допустимого, которые должны соответствовать определенным критериям.

Если новая цена оканчивается цифрами, которые не превышают 50 центов (как в приведенном выше примере с ценой `$10.13`), то специалисты по маркетингу требуют увеличить эту цену, оставив то же значение в долларах, но указав в конце 49 центов (в данном примере цена должна быть увеличена до `$10.49`). Цены, оканчивающиеся значениями от 50 до 75 центов, должны быть откорректированы так, чтобы оканчивались на 75 центов, а цены, оканчивающиеся значением, превышающим 75 центов, должны быть откорректированы так, чтобы они оканчивались на 95 центов. Некоторые примеры того, как эти требования реализуются на практике, приведены в табл. 12.1.

Таблица 12.1. Примеры округления цен в соответствии с заданными требованиями

Новая цена	Новая цена после округления
`$10.13`	`$10.49`
`$17.57`	`$17.75`
`$27.75`	`$27.75`
`$79.99`	`$79.95`

С формальной точки зрения эти требования можно было бы выполнить с помощью вложенных операторов `IF...ELSE`, но против этого свидетельствуют приведенные ниже соображения.

- Сформированный на этой основе код стал бы гораздо более трудным для восприятия, особенно если бы реализуемые с его помощью правила были еще сложнее.
- Для реализации кода потребовалось бы применить курсор (а это — неудачное решение!) и обрабатывать таблицу построчно.

В связи с этим указанное решение является неприемлемым.

Но применение оператора CASE позволяет значительно упростить реализацию предъявленных требований. Более того, этот оператор позволяет встраивать выражения проверки условий непосредственно в запрос и использовать в операции с множеством строк, а это почти всегда означает, что достигается гораздо более высокая производительность, чем при использовании курсоров.

Ознакомившись с таким предложением по организации обработки данных, специалисты из отдела маркетинга решили проверить, какие результаты будут получены после повышения цен на 10%, поэтому в оператор CASE было включено выражение, обеспечивающее прибавление к стоимости 10%-й наценки, и в конечном итоге после проведения небольшого дополнительного анализа был составлен сценарий, позволяющий решить поставленную задачу по округлению цен:

```
USE Northwind
GO
/* Объявления всех переменных, необходимых для хранения нормативных данных,
** сосредоточены в одном месте. Это позволяет упростить в дальнейшем
** корректировку запроса */
DECLARE @Markup     money
DECLARE @Multiplier money
SELECT @Markup = .10              -- Переменной @Markup присваивается
                                  -- значение наценки
SELECT @Multiplier = @Markup + 1 -- Необходимо определить конечную цену,
                                  -- а не величину повышения, поэтому к
                                  -- наценке прибавляется 1
/* Выполнение запроса для проверки полученных результатов. Необходимо отметить,
** что в данном случае в целях упрощения обработка ограничивается десятью
** максимальными значениями. Но на практике этого не следует делать. В крайнем
** случае можно применить более сложную конструкцию WHERE для повышения цен
** на товары определенной категории */
SELECT TOP 10 ProductID, ProductName, UnitPrice,
  UnitPrice * @Multiplier AS "Marked Up Price", "New Price" =
  CASE WHEN FLOOR(UnitPrice * @Multiplier + .24)
         > FLOOR(UnitPrice * @Multiplier)
               THEN FLOOR(UnitPrice * @Multiplier) + .95
       WHEN FLOOR(UnitPrice * @Multiplier + .5) >
           FLOOR(UnitPrice * @Multiplier)
               THEN FLOOR(UnitPrice * @Multiplier) + .75
       ELSE FLOOR(UnitPrice * @Multiplier) + .49
  END
FROM Products
ORDER BY ProductID DESC     -- Сортировка проводится по убыванию, поскольку
                            -- в противном случае наилучшие примеры
                            -- находились бы в конце
```

В этом примере используется весьма простая функция FLOOR, которая принимает переданное ей дробное значение и округляет его в меньшую сторону до ближайшего целого числа.

Опытным разработчикам часто приходится сталкиваться с тем, что заказчики не всегда могут сразу предусмотреть все необходимые проверки; это особенно характерно для специалистов в области маркетинга или торговли. В этом нет ничего предосудительного, ведь они гораздо лучше разбираются в своей работе, чем в программировании. Но в конечном итоге обнаруживается, что после проведения одной проверки

требуется еще одна, которая не предусматривалась ранее. Зная об этом, автор проявил предусмотрительность и учел такую возможность в своем сценарии; теперь, если окажется, что нужно проверить работу сценария на примере 15%-й наценки, то достаточно будет изменить значение, применяемое для инициализации переменной @Markup. Но все же на этот раз рассмотрим, какие результаты будут получены при использовании 10%-й наценки:

```
ProductID   ProductName                 UnitPrice   Marked Up Price  New Price
---------   --------------------------  ---------   ---------------  ---------
77          Original Frankfurter grune So?e  13.0000     14.3000        14.4900
76          Lakkalikoori                18.0000     19.8000        19.9500
75          Rhonbrau Klosterbier         7.7500      8.5250         8.7500
74          Longlife Tofu               10.0000     11.0000        11.4900
73          Rod Kaviar                  15.0000     16.5000        16.7500
72          Mozzarella di Giovanni      34.8000     38.2800        38.4900
71          Flotemysost                 21.5000     23.6500        23.7500
70          Outback Lager               15.0000     16.5000        16.7500
69          Gudbrandsdalsost            36.0000     39.6000        39.7500
68          Scottish Longbreads         12.5000     13.7500        13.7500
(10 row(s) affected)
```

Анализ этих результатов показывает, что они полностью соответствуют предъявленным требованиям. Более того, для получения этих данных не пришлось применять курсор.

Теперь рассмотрим, как включить оператор CASE, который был описан в последнем примере, в хранимую процедуру, которую могли бы вызывать сами специалисты отдела маркетинга.

Прежде чем приступать к преобразованию кода, подобного описанному выше, в хранимую процедуру, необходимо определить, какая информация должна быть задана при каждом вызове хранимой процедуры. В настоящем примере достаточно лишь указать величину наценки в процентах. Это означает, что хранимая процедура должна иметь только один параметр (величину наценки в процентах), а другие переменные хранимой процедуры могут быть оформлены как внутренние.

Таким образом, чтобы внести изменения в данный конкретный сценарий, достаточно лишь преобразовать одну переменную в параметр и ввести оператор CREATE. Тем не менее предусмотрим еще одно изменение — немного откорректируем комментарии:

```
USE Northwind
GO

CREATE PROC spMarkupTest
   @MarkupAsPercent   money
AS
   DECLARE @Multiplier money
-- Необходимо определить конечную цену, а не величину повышения
SELECT @Multiplier = @MarkupAsPercent / 100 + 1 /* Значение вычисляется с учетом
                                                ** величины повышения, поэтому
                                                ** необходимо прибавить единицу

** Выполнение запроса для проверки полученных результатов. Необходимо отметить,
** что в данном случае в целях упрощения обработка ограничивается десятью
** максимальными значениями. Но на практике этого не следует делать. В крайнем
```

```
** случае можно применить более сложную конструкцию WHERE для повышения цен на
** товары определенной категории */
SELECT TOP 10 ProductId, ProductName, UnitPrice,
  UnitPrice * @Multiplier AS "Marked Up Price", "New Price" =
  CASE WHEN FLOOR(UnitPrice * @Multiplier + .24)
          > FLOOR(UnitPrice * @Multiplier)
                THEN FLOOR(UnitPrice * @Multiplier) + .95
       WHEN FLOOR(UnitPrice  * @Multiplier + .5) >
            FLOOR(UnitPrice * @Multiplier)
                THEN FLOOR(UnitPrice * @Multiplier) + .75
       ELSE FLOOR(UnitPrice * @Multiplier) + .49
  END
FROM Products
ORDER BY ProductID DESC     -- Сортировка проводится по убыванию, поскольку
                            -- в противном случае наилучшие примеры
                            -- находились бы в конце
```

После этого для вызова на выполнение хранимой процедуры достаточно ввести ее имя после команды EXEC и задать параметр:

```
EXEC spMarkupTest 10
```

Полученные при этом результаты должны полностью соответствовать тем, которые формируются при использовании кода в форме сценария. Тем не менее, представив код в форме хранимой процедуры, мы достигаем описанных ниже преимуществ.

- Упрощается использование кода для неопытных пользователей.
- Сокращается время обработки данных.

То, что использование кода для конечного пользователя значительно упрощается, вполне очевидно. Рядовые пользователи чаще всего чувствуют растерянность, приступая к использованию кода сценария, даже если им приходится корректировать только одну строку. А после оформления сценария в виде хранимой процедуры им приходится вводить всего лишь три ключевых слова, в том числе значение параметра.

А что касается повышения производительности, то при такой интерактивной организации работы, как в рассматриваемом примере, это повышение практически не имеет никакого значения. Но следует знать, что обработка данных при использовании хранимой процедуры, а не сценария немного ускоряется (во многих случаях повышение быстродействия измеряется всего лишь в миллисекунду, а в других случаях представляет собой более значительную величину). Дополнительная информация на эту тему приведена в конце настоящей главы.

Организация циклов с помощью оператора WHILE

Оператор WHILE в языке SQL действует во многом так же, как и в других языках, с которыми обычно приходится работать программисту. По сути в этом операторе до начала каждого прохода по циклу проверяется некоторое условие. Если перед очередным проходом по циклу проверка условия приводит к получению значения TRUE, осуществляется проход по циклу, в противном случае выполнение оператора завершается.

Оператор WHILE имеет следующий синтаксис:

```
WHILE <Boolean expression>
    <sql statement> |
```

```
[BEGIN
     <statement block>
     [BREAK]
     <sql statement> | <statement block>
     [CONTINUE]
END]
```

Безусловно, с помощью оператора WHILE можно обеспечить выполнение в цикле только одного оператора (по аналогии с тем, как обычно используется оператор IF), но на практике конструкции WHILE, за которыми не следует блок BEGIN...END, соответствующий полному формату оператора, встречаются редко.

Оператор BREAK позволяет выйти из цикла, не ожидая того, как будет выполнен проход до конца цикла и произойдет повторная проверка условного выражения.

Разумеется, автор — не первый, кто начал настаивать на этом, но использование оператора BREAK обычно рассматривается как признак плохого стиля программирования, о чем говорят даже классические учебники. Я также стараюсь всегда отстаивать эту точку зрения и избегать применения операторов разрыва цикла при любой возможности. Чаще всего удается действительно обойтись без операторов BREAK, поменяв местами несколько групп операторов и добившись тех же результатов. При этом обычно достигается также еще одно преимущество — повышение удобства кода для чтения. Дело в том, что следить за работой циклической структуры (или даже, вообще говоря, любой программной структуры) намного легче, если в ней имеется единственная точка входа и единственная точка выхода. А если используется оператор BREAK, такой принцип организации кода нарушается.

Но несмотря на сказанное, иногда действительно структура кода становится хуже после его переформатирования в целях исключения оператора BREAK. Кроме того, мне иногда приходилось сталкиваться с тем, что программисты были даже готовы использовать код, обладающий гораздо более низким быстродействием, лишь бы не применять оператор BREAK. Но это, безусловно, не рекомендуется.

Оператор CONTINUE в определенном смысле является полностью противоположным оператору BREAK. Кратко можно описать действие оператора CONTINUE так, что он обеспечивает переход в начало цикла WHILE. Сразу после обнаружения оператора CONTINUE в цикле, независимо от того, где он находится, происходит переход в начало цикла и повторное вычисление условного выражения (а если значение этого выражения больше не равно TRUE, осуществляется выход из цикла).

Рассмотрим небольшой пример, чтобы ознакомиться с тем, как работает цикл WHILE. Как уже было сказано выше, цикл WHILE редко применяется не в сочетании с курсором, поэтому автору пришлось поневоле составить довольно надуманный пример.

В данном случае создается нечто вроде процесса мониторинга. Для этого используются цикл WHILE и оператор WAITFOR (более подробное описание оператора WAITFOR приведено в следующем разделе). В данном примере создается процедура автоматического обновления статистических данных один раз в сутки:

```
WHILE 1 = 1
BEGIN
   WAITFOR TIME '01:00'
   EXEC sp_updatestats
   RAISERROR('Statistics Updated for Database', 1, 1) WITH LOG
END
```

С помощью этой процедуры через каждые сутки в один час ночи обновляются статистические данные по каждой таблице в базе данных и журнальная запись с информацией об этом факте вносится и в журнал SQL Server, и в журнал приложений системы Windows NT. Чтобы убедиться в том, что приведенный здесь сценарий действует правильно, оставьте его работать на ночь, а утром проверьте журналы.

Следует отметить, что при обычных обстоятельствах планирование заданий по принципу использования подобного бесконечного цикла не рекомендуется. Если есть необходимость в том, чтобы какой-то сценарий вызывался на выполнение через каждые сутки, установите задание с помощью программы Management Studio. Это позволяет не только избавиться от необходимости поддерживать постоянно открытым одно из соединений (с чем связано выполнение приведенного выше примера), но и получить возможность предусмотреть выполнение последующих действий с учетом успешного или неудачного завершения сценария. Кроме того, можно организовать отправку сообщений по электронной почте или по сети, указывая в них информацию о состоянии завершения сценария.

Оператор WAITFOR

В процессе работы часто возникают такие ситуации, что некоторое действие нежелательно или невозможно выполнить прямо сейчас, но не хочется также ждать, когда наступит время, подходящее для его выполнения.

В подобных ситуациях можно воспользоваться оператором WAITFOR и поручить СУБД SQL Server обеспечить необходимое ожидание. Оператор WAITFOR имеет очень простой синтаксис:

```
WAITFOR
   DELAY <'time'> | TIME <'time'>
```

Оператор WAITFOR осуществляет именно то, о чем говорит его имя, — ожидает наступления того момента, который указан в параметре его вызова. Может быть указано либо явно заданное время суток, в которое с помощью этого оператора должно быть осуществлено определенное действие, либо указан промежуток времени, в течение которого необходимо организовать ожидание.

Параметр DELAY

Параметр DELAY позволяет указать промежуток времени, в течение которого необходимо организовать ожидание. С помощью этого параметра нельзя указать промежуток времени, превышающий одни сутки, поскольку в нем разрешается задавать только время в часах, минутах и секундах. Максимально допустимая продолжительность задержки равна 24 часам. Поэтому, например, если в сценарий включен приведенный ниже оператор, то будет выполнен весь код, предшествующий оператору WAITFOR, затем будет вызван на выполнение оператор WAITFOR, произведена задержка на один час, после чего выполнение кода продолжится с того оператора, который следует за ним.

```
WAITFOR DELAY '01:00'
```

Параметр TIME

Параметр TIME обеспечивает ожидание до конкретного времени суток. И в этом случае невозможно указать какую-либо дату; допускается только указывать время в 24-часовом формате. И в этом случае продолжительность ожидания, установленно-

го с помощью оператора WAITFOR, не превышает одних суток. Поэтому, например, если в сценарий включен приведенный ниже оператор, то будет выполнен весь код, предшествующий оператору WAITFOR, затем будет вызван на выполнение оператор WAITFOR, произведена задержка до одного часа ночи (01:00), после чего выполнение кода продолжится с того оператора, который следует за ним.

```
WAITFOR TIME '01:00'
```

Блоки TRY и CATCH

Полное описание данной темы будет приведено в одном из следующих разделов, посвященном обработке ошибок. Тем не менее эти программные конструкции относятся к категории средств управления процессом выполнения, поэтому должны быть также кратко описаны в данном контексте. Блоки TRY и CATCH впервые были введены в версии SQL Server 2005. Для тех, кто владеет только такими языками программирования, в которых не используются блоки TRY и CATCH, необходимо дать предварительное пояснение перед предстоящим обсуждением, что эти блоки предназначены только для обработки исключительных ситуаций.

В настоящем разделе будут приведены лишь самые основные сведения о блоках TRY и CATCH, в том числе описано их назначение. Кратко можно отметить, что в блок TRY помещается код, который в обычных обстоятельствах выполняется без возникновения каких-либо исключительных ситуаций, т.е. степень серьезности ошибок (о чем будет подробнее сказано немного позже) не превышает 10. А в блоке CATCH должен находиться код, который немедленно вызывается на выполнение, начиная с первой строки в этом блоке, и продолжается дальше, если в коде, находящемся в блоке TRY, возникает ошибка со степенью серьезности, превышающей 10 (равной 11 или больше).

> Следует отметить, что блок CATCH выполняется, только если ошибка не относится к такой категории, из-за которой немедленно прекращается выполнение всего сценария. Еще раз отметим, что эта тема будет рассматриваться более подробно в разделе, посвященном обработке ошибок, но достаточно сказать, что из-за ошибок некоторых типов выполнение хранимой процедуры немедленно прекращается; в данном случае это означает, что не будет даже выполнен блок CATCH.

Подтверждение успешного или неудачного завершения работы с помощью возвращаемых значений

Способы использования возвращаемых значений являются довольно разнообразными. Прежде всего они используются для возврата фактических данных, таких как идентификационное значение или данные о количестве строк, затронутых хранимой процедурой (однако возврат количества строк теперь рассматривается как недопустимая практика программирования, сохранившаяся с тех времен, когда еще не сложились оптимальные подходы к работе). Вместо этого следует переходить к исполь-

зованию таких способов организации работы, в которых возвращаемые значения применяются с той целью, для которой они действительно предназначены, — определения состояния выполнения хранимой процедуры.

На основании сказанного может сложиться впечатление, что автор имеет свое мнение по поводу того, как должны использоваться возвращаемые значения, и это действительно так, поскольку такое определенное мнение у меня есть. Во время моей учебы считалось, что возвращаемые значения должны служить в качестве "уловки", позволяющей избежать необходимости использовать выходные параметры; по сути рекомендовалось применять их как способ "срезать угол". Но, к счастью, мне удалось пересмотреть тот подход, которому меня обучили. Проблема заключается в том, что, как обычно, "срезая углы", приходится оставлять за собой разрушения, а в данном случае разрушается нечто важное.

Использование возвращаемых значений в качестве средства возврата данных в вызывающую процедуру противоречит самому назначению кода возврата и исключает возможность формировать и отправлять назад в точку вызова безукоризненно объективные коды ошибок. Иными словами, этого не следует делать!

Возвращаемые значения предназначены исключительно для указания на успешное или неудачное завершение хранимой процедуры и позволяют даже обозначить степень или характер успеха или неудачи. Программисты, работающие на языке C, уже наметили довольно удобную стратегию использования возвращаемых значений. В языке C стало общепринятой практикой применение возвращаемого значения функции в качестве кода завершения, любые ненулевые значения которого указывают на возникновение определенного рода проблем. Ознакомление с применяемыми по умолчанию в СУБД SQL Server кодами возврата показывает, что и в этой системе соблюдаются такие же правила.

Способ использования оператора RETURN

В действительности любая процедура, вызываемая на выполнение из программы, возвращает значение, независимо от того, предусмотрен ли в ней возврат значения или нет. По умолчанию после успешного завершения процедуры СУБД SQL Server автоматически возвращает значение, равное нулю.

Чтобы передать некоторое возвращаемое значение из хранимой процедуры обратно в вызывающий код, достаточно применить оператор RETURN:

```
RETURN [<integer value to return>]
```

> Обратите внимание на то, что возвращаемое значение должно быть целочисленным.

По-видимому, одной из наиболее важных особенностей оператора RETURN является то, что его выполнение приводит к безусловному завершению работы и выходу из хранимой процедуры. Это означает, что, независимо от местонахождения оператора RETURN в коде хранимой процедуре, после его выполнения больше не будет выполнена ни одна строка кода.

Под безусловным завершением работы подразумевается, что действие, предусмотренное оператором RETURN, осуществляется независимо от того, в каком месте кода он обнаруживается. С другой стороны, допускается наличие в коде хранимой процедуры нескольких операторов RETURN, а выполнение этих операторов происходит,

только если к этому приводит обычная структура управления процессом выполнения кода. Но после того как в коде встречается оператор RETURN, назад дороги нет.

Рассмотрим, как изменяется подход к организации кода после включения в него оператора RETURN на примере очень простой проверочной хранимой процедуры:

```
USE Northwind
GO
CREATE PROC spTestReturns
AS
   DECLARE @MyMessage         varchar(50)
   DECLARE @MyOtherMessage    varchar(50)
   SELECT @MyMessage = 'Hi, it''s that line before the RETURN'
   PRINT @MyMessage
   RETURN
   SELECT @MyOtherMessage = 'Sorry, but we won''t get this far'
   PRINT @MyOtherMessage
RETURN
```

Но после создания хранимой процедуры необходимо предусмотреть небольшой сценарий, позволяющий проверить интересующие нас аспекты работы хранимой процедуры. В частности, интерес представляют приведенные ниже вопросы.

- Какой вывод осуществляется из хранимой процедуры.
- Какое значение возвращает оператор RETURN.

Для перехвата значения, возвращаемого оператором RETURN, необходимо предусмотреть в операторе EXEC присваивание этого значения переменной. Например, в следующем коде возвращаемое значение присваивается переменной @ReturnVal:

```
EXEC @ReturnVal = spMySproc
```

Теперь поместим этот оператор в более удобный сценарий для проверки хранимой процедуры:

```
DECLARE @Return int
EXEC @Return = spTestReturns
SELECT @Return
```

Этот сценарий является небольшим, но удобным. После вызова его на выполнение можно убедиться в том, что оператор RETURN действительно прекращает выполнение кода так, что больше не может быть выполнен ни один оператор:

```
Hi, it's that line before the RETURN
-----------
0
(1 row(s) affected)
```

Кроме того, получено возвращаемое значение хранимой процедуры, которое оказалось равным нулю. Обратите внимание на то, что было получено нулевое значение, даже несмотря на то, что в хранимой процедуре не задано какое-либо конкретное возвращаемое значение; дело в том, что по умолчанию возвращаемое значение всегда равно нулю.

Над этим следует серьезно задуматься, ведь если по умолчанию возвращаемое значение равно нулю, это означает, что применяемое по умолчанию возвращаемое значение свидетельствует также об отсутствии ошибок, а в этом заключается весьма серьезная опасность. Таким об-

разом, важный вывод из сказанного состоит в том, что следует всегда явно задавать возвращаемые значения. Это позволяет безусловно иметь полные основания рассчитывать на то, что возвращаемое значение получено в соответствии с вашим замыслом, а не сформировано случайно.

На данном этапе ради интереса модифицируем эту хранимую процедуру, чтобы убедиться в том, что она позволяет передать в вызывающий код в качестве возвращаемого значения любые целочисленные значения:

```
USE Northwind
GO

ALTER PROC spTestReturns
AS
   DECLARE @MyMessage         varchar(50)
   DECLARE @MyOtherMessage    varchar(50)
   SELECT @MyMessage = 'Hi, it''s that line before the RETURN'
   PRINT @MyMessage
   RETURN 100
   SELECT @MyOtherMessage = 'Sorry, but we won''t get this far'
   PRINT @MyOtherMessage
RETURN
```

После повторного вызова на выполнение этого проверочного сценария обнаруживаются те же результаты, если не считать того, что изменилось возвращаемое значение:

```
Hi, it's that line before the RETURN
-----------
100
(1 row(s) affected)
```

Обработка ошибок

Безусловно, хотелось бы обойтись без этого раздела, чтобы в коде никогда не возникали ошибки и все приложения функционировали правильно. Но, к сожалению, об этом можно лишь мечтать, поэтому вернемся к реальности. Необходимо всегда быть готовым к тому, что в любой, даже самой замечательной программе может скрываться ошибка. К счастью, у нас есть возможность справляться с такими ошибками, хотя предназначенные для этой цели инструментальные средства иногда далеки от идеала. Тем не менее определенные способы позволяют в полной мере воспользоваться средствами, находящимися в вашем распоряжении, а также скомпенсировать недостатки средств обработки ошибок, предусмотренных в СУБД с поддержкой SQL.

Ошибки, которые могут обнаруживаться в процессе функционирования СУБД SQL Server, подразделяются на три описанных ниже основных типа.

- Ошибки, под воздействием которых возникают аварийные нарушения на этапе прогона программы и дальнейшее выполнение кода становится невозможным.
- Ошибки, которые обнаруживаются в СУБД SQL Server, но в связи с этим не возникают такие нарушения на этапе прогона программы, чтобы произошло прекращение выполнения кода. Ошибки указанного типа можно было бы назвать "побочными" ошибками.

- Ошибки, в большей степени обусловленные неправильной реализацией алгоритмов в приложении, которые применительно к СУБД SQL Server по сути не рассматриваются как нарушение в работе.

Очевидно, что тематика, касающаяся обработки ошибок, является очень сложной. Кроме того, с выпуском каждой версии SQL Server происходят важные изменения в подходах к обработке ошибок, о чем мы постараемся рассказать в данном разделе.

Как уже было отмечено в настоящей книге, литература, посвященная версии SQL Server 2005, еще весьма немногочисленна, но я предполагаю, что новые издания с описанием этой версии, которые появятся на рынке, не будут содержать большой объем материала, касающегося предыдущих версий. Я также в основном стремился не перегружать изложение материала лишними подробностями, чтобы не создавать ненужных сложностей. Несмотря на сказанное, в настоящем разделе будет часто идти речь о предыдущих версиях. Дело в том, что большинству разработчиков программного обеспечения для баз данных приходится либо самим работать с предыдущими версиями, либо эксплуатировать код, относящийся к версиям, предшествующим SQL Server 2005. С точки зрения изложенного в настоящем разделе, это очень важно, поскольку в SQL Server 2000 и в предыдущих версиях формально не было предусмотрено использование обработчиков ошибок.

С учетом этого я намереваюсь представить в данном разделе краткое описание методов обработки ошибок, применявшихся в предыдущих версиях. Это будет сделано с единственной целью — помочь читателю понять, почему в том коде, который относится к старым версиям, обработка ошибок организована именно так. Если же читатель уверен в том, что ему никогда не придется сталкиваться с унаследованным кодом и он имеет возможность заниматься разработкой программного обеспечения только для версии SQL Server 2005, то он может непосредственно перейти к разделам с описанием средств обработки ошибок, основанных на использовании конструкций `TRY` *и* `CATCH`.

В версии SQL Server 2005 предусмотрена новая модель обработки ошибок, но в ней сохранился один компонент, характерный для предыдущих версий, — средства обработки ошибок этапа прогона высокого уровня.

Эти средства позволяют вырабатывать сообщения об ошибках, которые вызывают немедленное прекращение выполнения сценария СУБД SQL Server. Применение этих средств было возможно до введения конструкций `TRY` и `CATCH` и остается возможным и после их введения. В самой СУБД SQL Server не всегда возможна выработка сообщений об ошибках, имеющих достаточную степень серьезности, чтобы вызвать появление ошибки этапа прогона. Безусловно, новые конструкции `TRY` и `CATCH` позволяют реализовывать более разносторонние методы обработки некоторых ошибок по сравнению с применявшимися ранее, но все равно иногда даже не представляется возможным передать в хранимую процедуру информацию о том, что произошла какая-то ошибка. С другой стороны, все современные объектные модели доступа к данным обеспечивают передачу сообщений о подобных ошибках, поэтому эти сведения можно получить в клиентском приложении и предпринять какие-то меры по исправлению возникших ошибок.

Применявшиеся ранее методы обработки ошибок

В версиях, предшествующих SQL Server 2005, с формальной точки зрения не было предусмотрено использование обработчиков ошибок и в связи с этим не было возможности вызывать на выполнение какой-то код при обнаружении определенной

ошибки. Вместо этого приходилось контролировать условия возникновения ошибок непосредственно в коде и при обнаружении ошибки предпринимать действия по ее исправлению (причем, возможно, по истечении значительного времени после того, как фактически произошла ошибка).

Обработка побочных ошибок

Побочные ошибки представляют собой те небольшие, но неприятные нарушения в работе, из-за которых функционирование СУБД SQL Server продолжается, как прежде, но по каким-то причинам намеченные действия оканчиваются неудачей. Например, попытаемся вставить в таблицу `Order Details` такую строку, которая не имеет соответствующей ей строки в таблице `Orders`:

```
USE Northwind
GO
INSERT INTO [Order Details]
   (OrderID, ProductID, UnitPrice, Quantity, Discount)
VALUES
   (999999,11,10.00,10, 0)
```

СУБД SQL Server не выполнит эту операцию вставки, поскольку на таблице `Order Details` задано ограничение `FOREIGN KEY`, которое ссылается на первичный ключ в таблице `Orders`. В данном случае в таблице `Orders` отсутствует строка с идентификатором `OrderID`, равным `999999`, поэтому строка, которую мы пытаемся вставить в таблицу `Order Details`, нарушает ограничение внешнего ключа и отвергается:

```
Msg 547, Level 16, State 0, Line 2
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Order_Details_Orders". The conflict occurred in database "Northwind",
table "Orders", column 'OrderID'.
The statement has been terminated.
```

Обратите внимание на то, что сообщение об ошибке, сформированное в данном случае, имеет определенный код — 547, и этим можно воспользоваться для обработки ошибок.

Использование системной переменной @@ERROR

Речь о данной конкретной системной переменной `@@ERROR` уже шла при описании средств создания сценариев, а в этом разделе будет описан способ ее практического применения.

Напомним, что системная переменная `@@ERROR` содержит номер ошибки, обнаруженной при выполнении последнего по времени оператора T-SQL. Если значение этой переменной равно нулю, то ошибка не была обнаружена.

> При использовании `@@ERROR` необходимо учитывать одно предостережение — значение этой системной переменной переустанавливается после выполнения каждого нового оператора. Это означает, что если требуется отложить анализ текущего значения или воспользоваться им в нескольких местах кода, то необходимо перенести его на хранение в какое-то другое место, например, в локальную переменную, предназначенную специально для этой цели.

Воспользуемся значением @@ERROR, полученным при выполнении приведенного выше оператора INSERT:

```
USE Northwind
GO
DECLARE  @Error  int
-- Фиктивный оператор INSERT - в базе данных Northwind отсутствует значение
-- OrderID, равное 999999
INSERT INTO [Order Details]
   (OrderID, ProductID, UnitPrice, Quantity, Discount)
VALUES
   (999999,11,10.00,10, 0)

-- Сохранение значения кода ошибки. Следует отметить, что после выполнения этого
-- оператора переменной @@Error будет присвоено соответствующее ему значение
-- кода ошибки
SELECT @Error = @@ERROR
-- Вывод пустой разделительной строки
PRINT ''
-- Значение переменной, предназначенной для хранения информации, является таким,
-- как и следовало ожидать
PRINT 'The Value of @Error is ' + CONVERT(varchar, @Error)
-- Значение переменной @@ERROR переустановлено и снова стало равным нулю
PRINT 'The Value of @@ERROR is ' + CONVERT(varchar, @@ERROR)
```

После этого вызовем на выполнение подготовленный сценарий и проверим, как изменяется значение @@ERROR:

```
Msg 547, Level 16, State 0, Line 5
The INSERT statement conflicted with the FOREIGN KEY constraint
"FK_Order_Details_Orders". The conflict occurred in database "Northwind",
table "Orders", column 'OrderID'.
The statement has been terminated.
The Value of @Error is 547
The Value of @@ERROR is 0
```

Эти результаты весьма наглядно демонстрируют, насколько важно сохранить текущее значение @@ERROR. Первое сообщение об ошибке является по своему характеру исключительно информационным. СУБД SQL Server активизирует эту ошибку, но не завершает аварийно выполнение кода. В действительности единственной частью данного сообщения, к которой может быть получен доступ к хранимой процедуре, является номер ошибки. Но этот номер ошибки сохраняется в системной переменной @@ERROR лишь до вызова на выполнение следующего оператора T-SQL, а после этого исчезает.

> Обратите внимание на то, что @Error и @@ERROR — две отдельные и различимые переменные, которые могут использоваться в коде независимо друг от друга. Это обусловлено не только тем, что имена переменных отличаются регистром букв (но чувствительность к регистру в именах переменных может зависеть от настройки конфигурации сервера), а, скорее, различием в том, какую область определения имеет та и другая переменная. В состав первого и второго имени входят соответственно символы @ и @@, поэтому для проведения различий между указанными именами достаточно лишь того, что они содержат в качестве префикса разное количество символов @.

Использование системной переменной @@ERROR в хранимой процедуре

Снова рассмотрим хранимую процедуру spInsertDateValidatedOrder, о которой шла речь при описании операторов IF...ELSE. Во всех приведенных выше примерах функционирование этой хранимой процедуры происходило просто безукоризненно. Иного и не следовало ожидать, поскольку все примеры были тщательно подготовлены. Тем не менее в реальном мире не всегда все происходит так же гладко. В действительности трудно даже представить себе, что может пользователь передать на обработку в подготовленный вами код. История хранит печальные воспоминания о тех программистах, которые надеялись на то, что ими учтены все возможные ситуации, но в первые же минуты после передачи в эксплуатацию своих программ обнаруживали, что пользователям уже удалось нарушить все возможные процедуры обработки данных (очевидно, что те ситуации, которые действительно возникли, на самом деле не были учтены).

Чтобы нарушить работу рассматриваемой хранимой процедуры, достаточно лишь внести небольшое изменение в проверочный сценарий:

```
USE Northwind
GO
DECLARE   @MyIdent   int
DECLARE   @MyDate    smalldatetime
SELECT @MyDate = GETDATE()
EXEC spInsertDateValidatedOrder
   @CustomerID = 'ZXZXZ',
   @EmployeeID = 5,
   @OrderDate  = @MyDate,
   @ShipVia    = 3,
   @Freight    = 5.00,
   @OrderID    = @MyIdent OUTPUT
SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM Orders
WHERE OrderID = @MyIdent
```

Это с виду незначительное изменение приводит к полному прекращению работы хранимой процедуры:

```
The Time of Day in Order Date was truncated
Server: Msg 547, Level 16, State 1, Procedure spInsertDateValidatedOrder,
Line 44
INSERT statement conflicted with COLUMN FOREIGN KEY constraint
'FK_Orders_Customers'. The conflict occurred in database 'Northwind',
table 'Customers', column 'CustomerID'.
The statement has been terminated.
OrderID     CustomerID     EmployeeID     OrderDate      ShipName
---------   ------------   -----------    ---------      --------------
(0 row(s) affected)
```

Вставка подготовленной строки не была выполнена. И этого не могло произойти, ведь для этого и существуют ограничения — чтобы предотвратить вставку в базу данных строк, не соответствующих требованиям.

Но неприятным следствием возникающей при этом ситуации становится появление многословного и пугающего сообщения, которое практически недоступно для

понимания среднего пользователя. Поэтому в сценарии необходимо предусмотреть проверку значения @@ERROR и передачу пользователю соответствующего сообщения.

Такую задачу можно легко решить, используя оператор IF...ELSE либо с системной переменной @@ERROR (если есть возможность немедленно проверить значение этой переменной и эту проверку достаточно провести только один раз), либо с локальной переменной, в которую перед этим было перенесено значение @@ERROR.

Сам автор стремится к созданию единообразного кода. Поэтому я всегда переношу значение системной переменной в локальную переменную и все предусмотренные проверки провожу с использованием этой переменной, даже если такую проверку требуется выполнить только один раз. Но я должен согласиться с тем, что, придерживаясь такого подхода, принадлежу к меньшинству. В связи с созданием в общем-то ненужной переменной немного увеличивается потребление памяти (ведь без такой промежуточной переменной можно было бы обойтись), кроме того, требуется дополнительный оператор присваивания (чтобы перенести значение @@ERROR в локальную переменную). Безусловно, оба эти фактора способствуют увеличению объема потребляемых программой ресурсов, но они чрезвычайно малы, поэтому я с успехом пропагандирую свой подход среди программистов, которые сами убеждаются, читая мой код, насколько приятно знать, что одни и те же действия неизменно реализуются с помощью одного и того же способа.

Кроме того, нет особого смысла дополнительно предусматривать выборку уже вставленной строки, поэтому мы можем пропустить соответствующую часть сценария, не имеющую отношения к рассматриваемой теме.

Таким образом, внесем ряд изменений, имеющих отношение к рассматриваемой проблеме обеспечения ссылочной целостности, и исключим код, который не относится к этой ситуации возникновения ошибок:

```
USE Northwind
GO
ALTER PROC spInsertDateValidatedOrder
  @CustomerID       nvarchar(5),
  @EmployeeID       int,
  @OrderDate        datetime     = NULL,
  @RequiredDate     datetime     = NULL,
  @ShippedDate      datetime     = NULL,
  @ShipVia          int,
  @Freight          money,
  @ShipName         nvarchar(40) = NULL,
  @ShipAddress      nvarchar(60) = NULL,
  @ShipCity         nvarchar(15) = NULL,
  @ShipRegion       nvarchar(15) = NULL,
  @ShipPostalCode   nvarchar(10) = NULL,
  @ShipCountry      nvarchar(15) = NULL,
  @OrderID          int     OUTPUT
AS

-- Объявления переменных
DECLARE   @Error                int
DECLARE   @InsertedOrderDate    smalldatetime
/* Проводится проверка для определения того, не отстоит ли дата поставки больше
** чем на семь дней от текущей даты и в случае положительного ответа эта дата
** заменяется NULL-значением. В противном случае устанавливается значение времени,
** соответствующее полуночи */
```

```
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7
BEGIN
   SELECT @InsertedOrderDate = NULL
   PRINT 'Invalid Order Date'
   PRINT 'Supplied OrderDate was greater than 7 days old.'
   PRINT 'The value has been reset to NULL'
END
ELSE
BEGIN
   SELECT @InsertedOrderDate =
      CONVERT(datetime,(CONVERT(varchar,@OrderDate,112)))
      PRINT 'The Time of Day in Order Date was truncated'
END
/* Создание новой записи */
INSERT INTO Orders
VALUES
(
   @CustomerID,
   @EmployeeID,
   @InsertedOrderDate,
   @RequiredDate,
   @ShippedDate,
   @ShipVia,
   @Freight,
   @ShipName,
   @ShipAddress,
   @ShipCity,
   @ShipRegion,
   @ShipPostalCode,
   @ShipCountry
)

-- Перенести код ошибки в локальную переменную и провести проверку
-- на наличие условия ошибки
SELECT @Error = @@ERROR
IF @Error != 0
BEGIN
    -- Возникла какая-то ошибка
   IF @Error = 547
   -- Проблема состоит в том, что нарушено ограничение. Вывести определенную
   -- справочную информацию, которая могла бы помочь пользователю определить
   -- наиболее вероятную причину нарушения в работе
   BEGIN
      PRINT 'Supplied data violates data integrity rules'
      PRINT 'Check that the supplied customer number exists'
      PRINT 'in the system and try again'
   END
   ELSE
   -- Возникла непредвиденная ситуация; сообщить о том, что причины ошибки
   -- неизвестны, и вывести сведения о самой ошибке
   BEGIN
      PRINT 'An unknown error occurred. Contact your System Administrator'
      PRINT 'The error was number ' + CONVERT(varchar, @Error)
   END
```

```
    -- Несмотря на то, что возникла ошибка, необходимо передать данные о ней
    -- в вызывающий код, чтобы можно было обеспечить надлежащую обработку ошибки
    RETURN @Error
END

/* Значение идентификации передается из вновь созданной записи
** в выходную переменную */
SELECT @OrderID = @@IDENTITY

RETURN
```

Теперь необходимо снова вызвать на выполнение проверочный сценарий, но применявшийся ранее вариант не совсем подходит для испытания рассматриваемой хранимой процедуры, поскольку теперь нам требуется перехватывать возвращаемое значение, чтобы знать, что произошло. Кроме того, не нужно вызывать на выполнение запрос для выборки только что вставленной строки на тот случай, если операция вставки окончилась неудачей, поэтому в случае возникновения ошибки такую проверку также можно пропустить:

```
USE Northwind
GO
DECLARE   @MyIdent   int
DECLARE   @MyDate    smalldatetime
DECLARE   @Return    int

SELECT @MyDate = GETDATE()

EXEC @Return   = spInsertDateValidatedOrder
   @CustomerID = 'ZXZXZ',
   @EmployeeID = 5,
   @OrderDate  = @MyDate,
   @ShipVia    = 3,
   @Freight    = 5.00,
   @OrderID    = @MyIdent OUTPUT

IF @Return = 0
   SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
   FROM Orders
   WHERE OrderID = @MyIdent
ELSE
   PRINT 'Value Returned was ' + CONVERT(varchar, @Return)
```

В действительности внесенные изменения не столь значительны, поскольку изменения коснулись только пяти строк. Тем не менее поведение сценария в случае возникновения ошибки стало совсем другим. Вызовем на выполнение этот сценарий, чтобы убедиться в том, что полученные результаты отличаются от тех, которые сформировались до ввода в действие средств проверки на наличие ошибок:

```
The Time of Day in Order Date was truncated
Server: Msg 547, Level 16, State 1, Procedure spInsertDateValidatedOrder, Line 42
INSERT statement conflicted with COLUMN FOREIGN KEY constraint
'FK_Orders_Customers'. The conflict occurred in database 'Northwind',
table 'Customers', column 'CustomerID'.
The statement has been terminated.
Supplied data violates data integrity rules
```

```
Check that the supplied customer number exists
in the system and try again
Value Returned was 547
```

Разрабатывая приведенный выше код, мы исходили из предположения, что отсутствует возможность применять обработчики ошибок, предусмотренные в наши дни в большинстве языков программирования, но все равно сумели создать очень удобные средства обработки ошибок.

Блоки TRY/CATCH

В предыдущем разделе описан так называемый традиционный способ обработки ошибок. Безусловно, этот способ нельзя назвать слишком удобным, но до выхода версии SQL Server 2005 он был единственно возможным. Но даже в случае использования указанной версии приходится ограничиваться этим способом, предусматривающим постоянную проверку значения системной переменной @@ERROR, если требуется обеспечить обратную совместимость. Если применяется только версия SQL Server 2005, то появляется возможность обработки ошибок с помощью блока TRY/CATCH, который находит гораздо более широкое применение в программировании.

Блок TRY/CATCH, предусмотренный в программном обеспечении SQL Server, чрезвычайно напоминает аналогичную конструкцию, применяемую во многих других языках программирования (таких как C, C++, C#, Delphi и т.д.). Блок TRY/CATCH имеет следующий синтаксис:

```
BEGIN TRY
    { <sql statement(s)> }
END TRY
BEGIN CATCH
    { <sql statement(s)> }
END CATCH [ ; ]
```

Иными словами, при использовании конструкции TRY/CATCH в СУБД SQL Server предпринимается попытка выполнить операторы, вложенные в блок BEGIN...END, который относится к блоку TRY. В случае успешного выполнения всех операторов в блоке TRY происходит переход к операторам, которые следуют за конструкцией TRY/CATCH. В противном случае при возникновении состояния ошибки, характеризующегося наличием степени серьезности 11–19 (информация о том, какую степень серьезности могут иметь ошибки, будет приведена ниже), в СУБД SQL Server немедленно осуществляется выход из блока TRY и переход к выполнению операторов блока CATCH, начиная с первого оператора.

В качестве примера рассмотрим вариант хранимой процедуры spInsertDateValidatedOrder, описанной в предыдущем разделе, в которой вместо применявшегося ранее способа обработки ошибок применяется новый способ. Рассматриваемый здесь вариант отличается тем, что проверка на наличие ошибки в ходе выполнения программы больше не проводится, а вместо этого применяется оператор INSERT, вложенный в блок TRY/CATCH. В результате этого исключается необходимость неоднократно проверять вручную значение системной переменной @@ERROR, поскольку состояние ошибки будет активироваться автоматически при возникновении любой ошибки со степенью серьезности от 11 до 19:

```
...
...
ELSE
BEGIN
   SELECT @InsertedOrderDate =
      CONVERT(datetime,(CONVERT(varchar,@OrderDate,112)))
      PRINT 'The Time of Day in Order Date was truncated'
END

/* Применение блока TRY/CATCH */
BEGIN TRY

   /* Создание новой записи */
   INSERT INTO Orders
   VALUES
   (
      @CustomerID,
      @EmployeeID,
      @InsertedOrderDate,
      @RequiredDate,
      @ShippedDate,
      @ShipVia,
      @Freight,
      @ShipName,
      @ShipAddress,
      @ShipCity,
      @ShipRegion,
      @ShipPostalCode,
      @ShipCountry
   )
END TRY
BEGIN CATCH
   -- Возникла какая-то ошибка; попытаемся определить, известен ли
   -- способ ее исправления
   DECLARE @ErrorNo  int,
        @Severity    tinyint,
        @State       smallint,
        @LineNo      int,
        @Message     nvarchar(4000)
   SELECT
       @ErrorNo   = ERROR_NUMBER(),
       @Severity  = ERROR_SEVERITY(),
       @State     = ERROR_STATE(),
       @LineNo    = ERROR_LINE (),
       @Message   = ERROR_MESSAGE()

   IF @ErrorNo = 547
   -- Проблема состоит в том, что нарушено ограничение. Вывести определенную
   -- справочную информацию, которая могла бы помочь пользователю определить
   -- наиболее вероятную причину нарушения в работе
   BEGIN
     PRINT 'Supplied data violates data integrity rules'
     PRINT 'Check that the supplied customer number exists'
     PRINT 'in the system and try again'
   END
```

```
    ELSE
    -- Возникла непредвиденная ситуация; сообщить о том, что причины ошибки
    -- неизвестны, и вывести сведения о самой ошибке
    BEGIN
      PRINT 'An unknown error occurred. Contact your System Administrator'
      PRINT 'The error was number ' + CONVERT(varchar, @Error)
    END
    -- Несмотря на то, что возникла ошибка, необходимо передать данные о ней
    -- в вызывающий код, чтобы можно было обеспечить надлежащую обработку ошибки
    RETURN @Error

  END CATCH

  /* Значение идентификации передается из вновь созданной записи
  ** в выходную переменную */
  SELECT @OrderID = @@IDENTITY

  RETURN
```

Следует отметить, что в приведенном здесь коде часть операторов удалена ради сокращения объема, поэтому читатель должен самостоятельно дополнить код на основании примера, приведенного в предыдущем разделе, прежде чем приступать к его использованию.

Необходимо подчеркнуть, что в приведенном выше коде для обнаружения состояния ошибки использовались некоторые специальные функции. Общие сведения о некоторых системных функциях, применяемых для обработки ошибок, приведены в табл. 12.2.

Кроме того, заслуживает внимания то, что в рассматриваемом коде полученные значения сразу же присваиваются переменным для предотвращения их потери в ходе дальнейшего выполнения программы (поскольку после каждого следующего вызова функции возвращаются другие результаты).

Таблица 12.2. Специальные функции, применяемые для проверки

Функция	Возвращаемое значение
`ERROR_NUMBER()`	Фактический номер ошибки. Если это — системная ошибка, то ей соответствует одна из строк в таблице `sysmessages`; эта строка содержит часть информации, которая может быть также получена с помощью некоторых других функций, предназначенных для обработки ошибок (для ознакомления с текстом сообщения об ошибке можно воспользоваться таблицей `sys.messages`)
`ERROR_SEVERITY()`	Возвращаемое значение показывает степень серьезности ошибки (error severity). В документации *Books Online* иногда вместо термина "error severity" применяется устаревший термин "error level". Это связано с тем, что специалисты корпорации Microsoft еще не успели его заменить
`ERROR_STATE()`	При возникновении системных ошибок функция `ERROR_STATE()` всегда возвращает значение `1`. В приведенном выше примере возвращаемое значение этой функции присваивается переменной (в расчете на то, что когда-либо оно потребуется в других программах), но пока не используется. В дальнейшем, после перехода к более подробному описанию средств обработки ошибок, будет показано, как активизировать ошибки, определяемые пользователем, а до ознакомления с этим описанием читатель может использовать возвращаемую информацию о состоянии для определения того, в каком именно месте кода хранимой процедуры, функции или триггера возникла ошибка (в том случае, если одна и та же ошибка может обнаружиться в разных местах кода)

Окончание табл. 12.2

Функция	Возвращаемое значение
ERROR_PROCEDURE()	Возвращаемое значение относится только к хранимым процедурам, функциям и триггерам, поэтому функция ERROR_PROCEDURE() в предыдущем примере не использовалась. Функция возвращает имя процедуры, при выполнении которой возникла ошибка; это очень удобно, если применяются вложенные ваши процедуры, поскольку возможно, что обработка ошибки должна фактически осуществляться в процедуре, отличной от той, в которой возникла ошибка
ERROR_LINE()	Возвращаемое значение полностью соответствует имени этой функции — оно показывает номер строки, в которой возникла ошибка
ERROR_MESSAGE()	Текст сообщения об ошибке, соответствующий указанному номеру ошибки. Если номер ошибки обозначает системную ошибку, то возвращаемое значение ERROR_MESSAGE() совпадает с тем результатом, который может быть получен путем выборки из таблицы sys.messages. Если же номер ошибки обозначает ошибку, определяемую пользователем, то функция ERROR_MESSAGE() возвращает текст, заданный при вызове функции RAISERROR

В рассматриваемом примере используется известный номер ошибки, который соответствует ошибке, активизируемой в СУБД SQL Server при осуществлении попытки создать объект, который уже существует. Со всеми системными сообщениями об ошибках можно ознакомиться, выполнив их выборку из таблицы sys.messages.

С выходом каждой версии объем выходных данных, полученных в результате выборки из таблицы sys.messages*, становится все больше. Это особенно проявляется в версии SQL Server 2005. Становится даже трудно найти искомую информацию путем просмотра. Автор предлагает, возможно, не такое уж изящное, но довольно эффективное решение — искусственно активизировать искомую ошибку и определять, какой номер ошибки ей соответствует.*

Достаточно просто предусмотреть вызов на выполнение интересующего вас кода (в данном случае оператора CREATE) и обработку ошибки, если она действительно может возникнуть.

Обработка ошибок еще до того, как они происходят

Иногда возникают ошибки, эффективного способа определить появление которых не предусмотрено даже в СУБД SQL Server, не говоря уже о самом приложении. Поэтому подобные ошибки необходимо предотвращать еще до того, как они происходят. В связи с этим требуется обеспечить проверку на наличие таких ошибок и их устранение непосредственно в приложении.

Еще раз вернемся к основному примеру хранимой процедуры, который рассматривается в настоящей главе, и обратим свое внимание на некоторые бизнес-правила, которые имеют алгоритмический характер, но не всегда могут быть реализованы в программном обеспечении базы данных. Например, предположим, что в базе данных допускается наличие неопределенных значений (NULL), но настало такое время, что больше не допускается столь неограниченное их использование, как раньше, в частности, принято решение, что нельзя больше разрешать вводить NULL вместо даты заказа OrderDate. Безусловно, в базе данных все еще остаются строки с незаполненной датой заказа, поэтому невозможно внести изменения в определение столбца OrderDate на уровне таблицы, чтобы этот столбец больше не допускал наличия NULL-значений. Поэтому необходимо найти какой-то другой подход.

Прежде всего мы должны отредактировать используемую хранимую процедуру так, чтобы она больше не допускала ввод в столбец NULL-значений. На первый взгляд такая задача кажется несложной, поскольку достаточно лишь удалить предусмотренное по умолчанию NULL-значение для соответствующего параметра, не правда ли? Тем не менее при реализации такого подхода возникают две описанные ниже проблемы.

- Если не задано значение поля `OrderDate`, то СУБД SQL Server формирует сообщение об ошибке, но все еще позволяет пользователю явно задавать NULL-значение.
- Даже если пользователь игнорирует необходимость ввести значение даты заказа, полученное пользователем сообщение об ошибке ни о чем ему не говорит.

Эти проблемы можно обойти, фактически продолжая использовать предусмотренное по умолчанию NULL-значение для поля `OrderDate` в том виде, как оно было задано, но на сей раз предусмотрев проверку данных, введенных пользователем. Если значение поля `OrderDate` после ввода данных пользователем содержит NULL-значение, то можно сделать вывод, что пользователь либо не задал значение даты заказа, либо ввел NULL-значение (но ни то ни другое больше не допускается), поэтому мы можем принять соответствующие меры. Таким образом, используя этот новый подход, мы должны найти ответ на вопрос о том, как выполнить проверку на наличие NULL-значения в поле `OrderDate`. Ответ на этот вопрос достаточно прост — применим такой способ, как и в конструкциях `WHERE` запросов:

```
IF @OrderDate IS NULL
  <abort the INSERT and print a message>
```

Внесем изменения в хранимую процедуру, которая нам уже хорошо знакома:

```
USE Northwind
GO
ALTER PROC spInsertDateValidatedOrder
  @CustomerID       nvarchar(5),
  @EmployeeID       int,
  @OrderDate        datetime = NULL,
  @RequiredDate     datetime = NULL,
  @ShippedDate      datetime = NULL,
  @ShipVia          int,
  @Freight          money,
  @ShipName         nvarchar(40) = NULL,
  @ShipAddress      nvarchar(60) = NULL,
  @ShipCity         nvarchar(15) = NULL,
  @ShipRegion       nvarchar(15) = NULL,
  @ShipPostalCode   nvarchar(10) = NULL,
  @ShipCountry      nvarchar(15) = NULL,
  @OrderID          int     OUTPUT
AS
-- Объявления переменных
DECLARE  @Error              int
DECLARE  @InsertedOrderDate  smalldatetime
/* Объявления констант. В действительности в программном обеспечении SQL Server
** не предусмотрено применение констант в классическом смысле этого слова,
** поэтому вместо констант используются переменные. В результате код становится
** более удобным для чтения, особенно если приходится проводить сравнение со
** списком констант в клиентской программе */
```

```
DECLARE   @INVALIDDATE   int
/* Константы после объявления необходимо инициализировать. Следует отметить, что
** в программном обеспечении SQL Server игнорируются пробелы между именем
** переменной и знаком "=". А в данном случае показано, как можно использовать
** пробелы для выравнивания значений констант и повышения удобства чтения */
SELECT @INVALIDDATE = -1000
/* Провести проверку для определения того, не отстоит ли дата поставки больше чем
** на семь дней от текущей даты (такая дата считается недействительной). Кроме
** того, провести проверку на наличие NULL-значений. Если имеет место одна из
** указанных ситуаций, завершить выполнение хранимой процедуры и вывести
** сообщение об ошибке */
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7 OR @OrderDate IS NULL
BEGIN
   PRINT 'Invalid Order Date'
   PRINT 'Supplied Order Date was greater than 7 days old '
   PRINT 'or was NULL. Correct the date and resubmit.'
   RETURN @INVALIDDATE
END
-- Предыдущая часть сценария выполнена успешно, поэтому можно без опасений
-- продолжить работу
SELECT @InsertedOrderDate =
   CONVERT(datetime,(CONVERT(varchar,@OrderDate,112)))
   PRINT 'The Time of Day in Order Date was truncated'

/* Создание новой записи */
INSERT INTO Orders
VALUES
(
   @CustomerID,
   @EmployeeID,
   @InsertedOrderDate,
   @RequiredDate,
   @ShippedDate,
   @ShipVia,
   @Freight,
   @ShipName,
   @ShipAddress,
   @ShipCity,
   @ShipRegion,
   @ShipPostalCode,
   @ShipCountry
)
-- Код завершения можно перенести в локальную переменную и проверить на наличие
-- условия ошибки
SELECT @Error = @@ERROR
IF @Error != 0
BEGIN
   -- Возникла какая-то ошибка
   IF @Error = 547
   -- Проблема состоит в том, что нарушено ограничение. Вывести определенную
   -- справочную информацию, которая могла бы помочь пользователю определить
   -- наиболее вероятную причину нарушения в работе
   BEGIN
      PRINT 'Supplied data violates data integrity rules'
      PRINT 'Check that the supplied customer number exists'
```

```
      PRINT 'in the system and try again'
   END
   ELSE
   -- Возникла непредвиденная ситуация; сообщить о том, что причины ошибки
   -- неизвестны, и вывести сведения о самой ошибке
   BEGIN
      PRINT 'An unknown error occurred. Contact your System Administrator'
      PRINT 'The error was number ' + CONVERT(varchar, @Error)
   END
   -- Несмотря на то, что возникла ошибка, необходимо передать данные о ней
   -- в вызывающий код, чтобы можно было обеспечить надлежащую обработку
ошибки
   RETURN @Error
END
/* Значение идентификации передается из вновь созданной записи
** в выходную переменную */
SELECT @OrderID = @@IDENTITY

RETURN
```

В этой процедуре предусмотрена возможность исправления целого ряда ошибок, поэтому для ее проверки необходимо применить несколько различных способов. Прежде всего введем допустимый идентификатор заказчика и вызовем хранимую процедуру на выполнение. При условии, что эта проверка окончится успешно, можно перейти к следующему этапу — ввести недопустимую дату:

```
USE Northwind
GO
DECLARE   @MyIdent   int
DECLARE   @MyDate    smalldatetime
DECLARE   @Return    int

SELECT @MyDate = '1/1/1999'

EXEC @Return   = spInsertDateValidatedOrder
   @CustomerID = 'ALFKI',
   @EmployeeID = 5,
   @OrderDate  = @MyDate,
   @ShipVia    = 3,
   @Freight    = 5.00,
   @OrderID    = @MyIdent OUTPUT
IF @Return = 0
   SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
   FROM Orders
   WHERE OrderID = @MyIdent
ELSE
   PRINT 'Value Returned was ' + CONVERT(varchar, @Return)
```

На этот раз после вызова хранимой процедуры на выполнение формируется следующее сообщение об ошибке:

```
Invalid Order Date
Supplied Order Date was greater than 7 days old
or was NULL. Correct the date and resubmit.
Value Returned was -1000
```

Обратите внимание на то, что приведенное выше сообщение об ошибке сформировано не в СУБД SQL Server, поскольку код, предусмотренный в этом сценарии, с точки зрения его выполнения в СУБД остается вполне приемлемым. Таким образом, здесь показан удобный способ, с помощью которого мы получили возможность, используя клиентскую программу (допустим, написанную на VB, C++ или каком-то другом языке программирования), сверить полученное значение с заранее заданной константой (в данном случае -1000) и отправить весьма содержательное сообщение конечному пользователю.

Активизация сообщений об ошибках вручную

Иногда возникают такие ситуации, которые с точки зрения эксплуатации приложения рассматриваются как ошибки, а что касается самой СУБД SQL Server, не рассматриваются как таковые, но для обработки этих ошибок желательно применить механизмы самой СУБД. Например, предположим, что в предыдущем примере возврат значения -1000 является признаком возникновения ошибки и об этом передается сообщение пользователю. Но вместо этого хотелось бы, чтобы в клиентском приложении была активизирована ошибка этапа прогона, а сообщение о данной ошибке было бы применено для вызова обработчика ошибок и осуществления соответствующих действий. Для такой цели может использоваться оператор RAISERROR языка T-SQL, который имеет довольно простой синтаксис:

```
RAISERROR (<message ID | message string>, <severity>, <state>
[, <argument>
[,<...n>]] )
[WITH option[,...n]]
```

Отдельные компоненты этой синтаксической структуры описаны в следующих разделах.

Параметр с обозначением идентификатора сообщения или строки сообщения

Параметр с обозначением идентификатора сообщения или строки сообщения позволяет указать, какое сообщение должно быть передано в клиентскую программу.

Если используется идентификатор сообщения, то создается активизированное вручную сообщение об ошибке с заданным идентификатором и текстом, соответствующим идентификатору и находящимся в таблице sysmessages базы данных master.

Для просмотра списка заранее заданных сообщений СУБД SQL Server можно в любое время вызвать на выполнение оператор `SELECT * FROM master..sysMessages`. *В формируемый при этом список входят также все сообщения, введенные в систему вручную с помощью хранимой процедуры* `sp_addmessage` *или с использованием программы Enterprise Manager.*

Таким образом, чтобы воспользоваться идентификатором сообщения, необходимо передать его на постоянное хранение вместе с соответствующим сообщением в таблицу sysmessages. Но предусмотрена также возможность просто задать строку сообщения в форме произвольного текста. Например, оператор

```
RAISERROR ('Hi there, I''m an error', 1, 1)
```

вызывает формирование следующего довольно простого сообщения об ошибке:

```
Msg 50000, Level 1, State 50000
Hi there, I'm an error
```

Обратите внимание на то, что система присвоила этому сообщению идентификатор, равный 50000, даже несмотря на то, что он не был задан в предыдущем операторе RAISERROR. Этот идентификатор ошибки присваивается по умолчанию любому сообщению об ошибке, введенному непосредственно в операторе RAISERROR, а не извлеченному из таблицы sysmessages. Вместо этого применяемого по умолчанию идентификатора ошибки можно явно задать другой идентификатор с помощью опции WITH SETERROR.

Степень серьезности ошибки

Понятие степени серьезности ошибки (severity) входит в состав терминологии, связанной со многими серверами Windows. Степень серьезности ошибки является указанием на то, какие меры следует принимать с учетом этой ошибки. Система обозначений степеней серьезности ошибок в СУБД SQL Server требует внимательного изучения. Эти обозначения охватывают широкий спектр сообщений об ошибках, включая те, которые по существу являются информационными (со значениями степеней серьезности 1–18), считаются относящимися к системному уровню (19–25) и даже рассматриваются как катастрофические (20–25). При активизации ошибки со степенью серьезности 19 или выше (системный уровень) необходимо также задавать опцию WITH LOG. При возникновении ошибок со степенями серьезности 20 и выше автоматически завершается работа пользовательских соединений (следует учитывать, что при этом пользователи выражают бурное возмущение!).

Рассмотрим более подробно систему обозначений степеней серьезности, применяемую в СУБД SQL Server, которая, как уже было сказано, заслуживает внимательного изучения. Фактически действия, предпринимаемые в СУБД SQL Server, в зависимости от степени серьезности ошибки подразделяются на большее число категорий, чем в других серверах NT, причем даже в документации *Books Online* перечислены не все эти категории. В целом значения степеней серьезности подразделяются на шесть основных группировок, которые перечислены в табл. 12.3.

Обозначение состояния

Обозначение состояния (state) представляет собой произвольную величину. Этот параметр оператора RAISERROR был введен в действие с учетом того, что одна и та же ошибка может возникнуть в несколько местах кода. А параметр с обозначением состояния предоставляет возможность передать вместе с сообщением своего рода маркер участка кода, который показывает, где именно произошла ошибка.

Числа с обозначением состояния могут находиться в пределах от 1 до 127. Совместное проведение поиска причин нарушений в работе с техническими специалистами по сопровождению корпорации Microsoft наводит на мысль о том, что они обладают какими-то недоступными для посторонних сведениями о том, какой смысл имеют некоторые обозначения состояния. Как уже было сказано, при обращении за консультацией к специалистам корпорации Microsoft по поводу какой-то ошибки эти специалисты чаще всего просят сообщить информацию о состоянии и используют ее в своей работе.

Таблица 12.3. Классификация значений степеней серьезности сообщений об ошибках, применяемых в СУБД SQL Server

Степень серьезности ошибки	Описание
1–9	Чисто информационные сообщения, содержащие в тексте сообщения конкретный код ошибки. Независимо от того, какое обозначение состояния применяется в операторе RAISERROR (об этом речь пойдет позже), в конечном итоге происходит возврат в качестве номера ошибки одного и того же значения (причины этого в настоящей книге не рассматриваются)
10	Эти сообщения — также информационные, но не активизируют ошибку в клиентской программе и не предоставляют каких-либо конкретных сведений об ошибке, кроме текста самого сообщения об ошибке
11–16	При получении этих сообщений выполнение процедуры завершается и в клиентской программе активизируется ошибка. Начиная с этого момента в информации о состоянии будет содержаться то значение, которое было установлено до возникновения ошибки
17	Обычно не рекомендуется задавать это значение степени серьезности в приложении, поскольку оно предназначено для использования только в СУБД SQL Server. По существу это значение указывает, что в СУБД SQL Server исчерпаны ресурсы (например, в базе данных tempdb не осталось свободного места), поэтому запрос не может быть выполнен
18–19	Оба эти значения степеней серьезности соответствуют серьезным ошибкам, причем при возникновении этих ошибок подразумевается, что для устранения их основополагающих причин требуется вмешательство системного администратора. Для обработки ошибок со степенью серьезности 19 требуется задать опцию WITH LOG, а сама информация о событии, связанном с возникновением ошибки, регистрируется в журнале событий Event Log операционной системы Windows NT или Win2K, если используются эти семейства операционных систем
20–25	Значения степеней серьезности от 20 до 25 являются катастрофическими не только для приложений, но и для пользовательских соединений. По существу, ошибки с такими степенями серьезности являются неисправимыми. Соединение с базой данных закрывается. Как и по отношению к ошибкам со степенью серьезности 19, необходимо использовать опцию WITH LOG. Сообщения, связанные с подобными ошибками, регистрируются в журнале Event Log (если соблюдаются все прочие условия, от которых зависит регистрация)

Параметры оператора активизации ошибки

При формировании некоторых заранее определенных сообщений об ошибках допускается задавать параметры. Это позволяет формировать сообщения об ошибках динамически, более точно учитывая конкретный характер ошибки. Разработчик имеет также возможность форматировать сообщения об ошибках таким образом, чтобы при формировании этих сообщений учитывались параметры.

Для преобразования постоянного сообщения об ошибке в такое сообщение, которое допускает использование динамически формируемой информации, необходимо отформатировать одну из постоянных частей этого сообщения так, чтобы в нем оставалось место для определяемого параметрами раздела сообщения. Эту задачу можно решить с использованием меток-заполнителей. Программисты, работающие на языке C или C++, могут рассматривать метки-заполнители параметров как аналоги компонентов строки формата функции printf. А для программистов, работающих на языке, отличном от C, понятие меток-заполнителей может оказаться непривычным. Все метки-заполнители начинаются со знака % и содержат в себе закодированное обозна-

чение типа информации, которая передается для подстановки вместо них в текст сообщения. Перечень индикаторов типов меток-заполнителей приведен в табл. 12.4.

Таблица 12.4. Перечень индикаторов типов меток-заполнителей

Индикатор типа метки-заполнителя	Тип значения
`D`	Целое число со знаком; обратите внимание на то, что в документации *Books Online* указано также, что допускается вместо `D` использовать индикатор типа `i`, но на практике при использовании этого значения возникают нарушения в работе
`O`	Восьмеричное число без знака
`P`	Указатель
`S`	Строка
`U`	Целое число без знака
`X` или `x`	Шестнадцатеричное число без знака

Кроме того, как показано в табл. 12.5, предусмотрена возможность задавать в качестве префикса для любого из указанных в табл. 12.4 индикаторов меток-заполнителей некоторые дополнительные флажки и информацию о ширине поля вывода.

Таблица 12.5. Дополнительные флажки индикаторов меток-заполнителей

Флажок	Назначение
- (дефис или знак "минус")	Обеспечивает выравнивание по левому краю. Применение этого флажка имеет смысл, только если задана постоянная ширина поля
+ (знак "плюс")	Указывает на то, что перед положительным или отрицательным числовым значением должен быть показан его знак
`0`	Указывает на то, что часть поля, находящуюся слева от числового значения, ширина которого меньше ширины поля, необходимо заполнить нулями
# (знак "диез")	Относится только к восьмеричным и шестнадцатеричным значениям. Указывает на то, что должен использоваться соответствующий префикс (`0` или `0x`), в зависимости от того, является ли выводимое значение восьмеричным или шестнадцатеричным
' ' (пробел)	Указывает на то, что часть поля, находящуюся слева от положительного числового значения, ширина которого меньше ширины поля, необходимо заполнить нулями

Наконец, предусмотрена также возможность задать ширину поля вывода, указать точность и обозначить параметр как имеющий длинный или короткий формат представления, как описано ниже.

- Ширина. Задается путем указания целочисленного значения, определяющего ширину поля вывода параметризованного значения. Вместо обозначения ширины может быть также задан символ звездочки (*). В таком случае СУБД SQL Server автоматически определяет ширину в зависимости от указанного значения точности.
- Точность. Определяет максимальное количество позиций для вывода цифр числовых данных.

- Обозначение длинного или короткого формата числовых данных. Задается с использованием символа h (короткий формат) или I (длинный формат), если параметр относится к одному из числовых типов с фиксированным форматом, т.е. является десятичным, восьмеричным или шестнадцатеричным.

Рассмотрим применение меток-заполнителей на примере:

```
RAISERROR ("This is a sample parameterized %s, along with a zero
padding and a sign%+010d",1,1, "string", 12121)
```

После вызова на выполнение этого оператора формируется сообщение об ошибке, которое выглядит немного иначе по сравнению с текстом, заданным в кавычках:

```
Msg 50000, Level 1, State 50000
This is a sample parameterized string, along with a zero
padding and a sign+000012121
```

Дополнительные значения были вставлены по порядку вместо меток-заполнителей, причем последнее значение переформатировано в соответствии с заданным форматом.

Опции WITH <option>

В настоящее время предусмотрены три опции, которые можно применять при активизации сообщения об ошибке отдельно и в различных сочетаниях:

- WITH LOG;
- WITH SETERROR;
- WITH NOWAIT.

Опция WITH LOG

Опция WITH LOG сообщает СУБД SQL Server, что сообщение об ошибке должно быть внесено в журнал ошибок SQL Server и в журнал приложения NT (последний применяется только в инсталляциях операционной системы NT). Применение этой опции является обязательным в сочетании с сообщениями, имеющими степень серьезности 19 или выше.

Опция WITH SETERROR

По умолчанию при выполнении оператора RAISERROR системной переменной @@ERROR не присваивается идентификатор сформированного сообщения об ошибке. Вместо этого значение системной переменной @@ERROR фактически отражает результат успешного или неудачного выполнения оператора RAISERROR. Опция WITH SETERROR позволяет отменить такое действие, чтобы системной переменной @@ERROR присваивалось значение, равное идентификатору сообщения об ошибке.

Опция WITH NOWAIT

При использовании опции WITH NOWAIT клиентское приложение немедленно получает уведомление о возникновении ошибки.

Ввод в систему определяемых пользователем сообщений об ошибках

Для ввода в систему дополнительных сообщений об ошибках может применяться специальная системная хранимая процедура, sp_addmessage, которая имеет следующий синтаксис:

```
sp_addmessage [@msgnum =] <msg id>,
[@severity =] <severity>,
[@msgtext =] <'msg'>
[, [@lang =] <'language'>]
[, [@with_log =] [TRUE|FALSE]]
[, [@replace =] 'replace']
```

Все параметры этой процедуры имеют практически то же назначение, что и параметры оператора RAISERROR, за исключением того, что в последнем случае дополнительно введен параметр с обозначением языка и параметр REPLACE, а также внесены небольшие изменения в опцию WITH LOG.

Параметр @lang

Параметр @lang позволяет указать, к какому языку программирования относится данное сообщение. Возможность предусмотреть отдельную версию сообщения для любого языка, о поддержке которого указано в таблице syslanguages, является очень ценной.

Параметр @with_log

Параметр@with_log действует во многом аналогично опции WITH LOG, предусмотренной в операторе RAISERROR, поскольку если этому параметру присвоено значение TRUE, то сообщения после их активизации автоматически регистрируются и в журнале ошибок SQL Server, и в журнале приложения NT (последний журнал используется, только если СУБД SQL Server эксплуатируется в операционной системе NT). Единственный нюанс, который приходится при этом учитывать, состоит в том, что для указания необходимости регистрации сообщений данному параметру присваивается значение TRUE, а не используется опция WITH LOG.

Следует очень критически относиться к сказанному об этом в документации Books Online. При недостаточно внимательном прочтении описания данной темы в документации можно вполне интерпретировать сказанное так, что параметру @with_log следует присвоить значение строковой константы 'WITH_LOG', тогда как фактически должно быть присвоено значение TRUE. По-видимому, еще большую путаницу вызывает то, что параметр @replace на первый взгляд кажется почти аналогичным по способу его использования, но ему должно быть присвоено значение строковой константы, а не TRUE.

Параметр @replace

Если с помощью процедуры sp_addmessage осуществляется редактирование существующего сообщения, а не создание нового, то необходимо присвоить параметру @replace значение 'REPLACE'. А при попытке отредактировать текст существующего сообщения без применения этого параметра активизируется сообщение об ошибке.

Рекомендуем создать заранее заданный список дополнительных сообщений, предназначенный для применения во всех разрабатываемых приложениях, поскольку это позволяет значительно повысить степень многократного использования сообщений, а также, что еще более важно, значительно повысить удобство для чтения кода приложения. Достаточно представить себе, насколько это удобно, если в каждом из приложений для базы данных используется постоянный список определяемых пользователем кодов ошибок. В таком случае можно легко подготовить файл с константами (например, ресурс или включаемую библиотеку), который содержит список применяемых сообщений об ошибках. При таком подходе можно даже подготовить включаемую библиотеку, в которой предусмотрены универсальные средства обработки некоторых или даже всех ошибок. Короче говоря, если вы намереваетесь разработать большое количество приложений для СУБД SQL Server в одной и той же среде, то рассмотрите возможность создать список сообщений об ошибках, общий для всех разрабатываемых приложений.

Использование хранимой процедуры sp_addmessage

Как уже было сказано, с помощью хранимой процедуры sp_addmessage в базу данных можно ввести сообщения, активизируемые с помощью оператора RAISERROR, в основном таким же образом, как активизируются произвольные сообщения.

Рассмотрим пример добавления в базу данных собственного определяемого пользователем сообщения, с помощью которого формируется предупреждение о том, что приведенная дата заказа является недопустимой:

```
sp_addmessage
   @msgnum = 60000,
   @severity = 10,
   @msgtext = '%s is not a valid Order date.
Order date must be within 7 days of current date.'
```

После вызова на выполнение этой хранимой процедуры появляется подтверждение того, что в базу данных добавлено новое сообщение:

```
(1 row(s) affected)
```

При вызове на выполнение хранимой процедуры sp_addmessage сообщение фактически добавляется в таблицу sysmessages базы данных master, независимо от того, с какой базой данных вы в этот момент работаете. Такая особенность рассматриваемой хранимой процедуры является очень важной, поскольку после переноса базы данных на новый сервер необходимо снова вводить пользовательские сообщения в базу данных master этого нового сервера (пользовательские сообщения не переносятся из базы данных master применявшегося ранее сервера в базу данных master нового сервера автоматически). Поэтому автор настоятельно рекомендует хранить все определяемые пользователем сообщения в отдельном сценарии, чтобы их всегда можно было легко ввести в новую систему.

Кроме того, следует отметить, что определяемые пользователем сообщения можно легко добавлять и удалять с помощью программы Enterprise Manager (щелкните правой кнопкой мыши на обозначении сервера, а затем перейдите к команде меню All Tasks | Manage SQL Server messages). Безусловно, такой метод добавления сообщений является легким и удобным, но при его использовании становится сложнее создавать и проверять сценарии, рекомендации по внедрению которых даны в предыдущем абзаце. Иначе говоря, автор не рекомендует использовать для работы с определяемыми пользователем сообщениями программу Enterprise Manager.

Удаление существующего сообщения, определяемого пользователем

Чтобы удалить из базы данных одно из определяемых пользователем сообщений, достаточно применить системную процедуру sp_dropmessage, которая имеет следующий синтаксис:

```
sp_dropmessage <msg num>
```

Практическое применение средств обработки ошибок, описанных в предыдущих разделах

В настоящем разделе приведен пример использования различных средств обработки ошибок, которые рассматривались в предыдущих разделах.

Прежде всего, если вы уже предприняли попытку удалить вновь введенное сообщение об ошибке с идентификатором 60000 с помощью хранимой процедуры sp_dropmessage, то отмените это действие — снова введите это сообщение, чтобы иметь возможность использовать его в данном примере.

В ходе решения рассматриваемой задачи мы должны поднять разрабатываемую хранимую процедуру выше на один уровень сложности, снова откорректировав так, чтобы в ней успешно применялись все описанные выше новые средства обработки ошибок. После завершения этой работы появится возможность вырабатывать доступные для перехвата сообщения об ошибках этапа выполнения в клиентской программе, чтобы можно было в связи с этим предпринимать необходимые действия.

Для этого следует ввести в параметр вызова одного из операторов PRINT такое изменение, чтобы этот оператор мог применяться в сочетании с оператором RAISERROR:

```
USE Northwind
GO
ALTER PROC spInsertDateValidatedOrder
  @CustomerID       nvarchar(5),
  @EmployeeID       int,
  @OrderDate        datetime     = NULL,
  @RequiredDate     datetime     = NULL,
  @ShippedDate      datetime     = NULL,
  @ShipVia          int,
  @Freight          money,
  @ShipName         nvarchar(40) = NULL,
  @ShipAddress      nvarchar(60) = NULL,
  @ShipCity         nvarchar(15) = NULL,
  @ShipRegion       nvarchar(15) = NULL,
  @ShipPostalCode   nvarchar(10) = NULL,
  @ShipCountry      nvarchar(15) = NULL,
  @OrderID          int     OUTPUT
AS
-- Объявления переменных
DECLARE  @Error              int
DECLARE  @BadDate            varchar(12)
DECLARE  @InsertedOrderDate  smalldatetime
/* Провести проверку для определения того, не отстоит ли дата поставки больше
** чем на семь дней от текущей даты (такая дата считается недействительной).
** Кроме того, провести проверку на наличие NULL-значений. Если имеет место
** одна из указанных ситуаций, завершить выполнение хранимой процедуры и
** вывести сообщение об ошибке */
```

```
IF DATEDIFF(dd, @OrderDate, GETDATE()) > 7 OR @OrderDate IS NULL
BEGIN
   -- Значение RAISERROR не относится к типу данных даты, поэтому вначале
   -- необходимо его преобразовать
   SELECT @BadDate = CONVERT(varchar, @OrderDate)
   RAISERROR (60000,1,1, @BadDate) WITH SETERROR
   RETURN @@ERROR
END
-- Предыдущая часть сценария выполнена успешно, поэтому можно без опасений
-- продолжить работу
SELECT @InsertedOrderDate =
   CONVERT(datetime,(CONVERT(varchar,@OrderDate,112)))
   PRINT 'The Time of Day in Order Date was truncated'
/* Создание новой записи */
INSERT INTO Orders
VALUES
(
   @CustomerID,
   @EmployeeID,
   @InsertedOrderDate,
   @RequiredDate,
   @ShippedDate,
   @ShipVia,
   @Freight,
   @ShipName,
   @ShipAddress,
   @ShipCity,
   @ShipRegion,
   @ShipPostalCode,
   @ShipCountry
)
-- Код завершения можно перенести в локальную переменную и проверить на
-- наличие условия ошибки
SELECT @Error = @@ERROR
IF @Error != 0
BEGIN
   -- Возникла какая-то ошибка
   IF @Error = 547
   -- Проблема состоит в том, что нарушено ограничение. Вывести определенную
   -- справочную информацию, которая могла бы помочь пользователю определить
   -- наиболее вероятную причину нарушения в работе
   BEGIN
      PRINT 'Supplied data violates data integrity rules'
      PRINT 'Check that the supplied customer number exists'
      PRINT 'in the system and try again'
   END
   ELSE
   -- Возникла непредвиденная ситуация; сообщить о том, что причины ошибки
   -- неизвестны, и вывести сведения о самой ошибке
   BEGIN
      PRINT 'An unknown error occurred. Contact your System Administrator'
      PRINT 'The error was number ' + CONVERT(varchar, @Error)
   END
   -- Несмотря на то, что возникла ошибка, необходимо передать данные о ней
   -- в вызывающий код, чтобы можно было обеспечить надлежащую обработку ошибки
   RETURN @Error
END
/* Значение идентификации передается из вновь созданной записи
** в выходную переменную */
SELECT @OrderID = @@IDENTITY
RETURN
```

Возможности, предоставляемые хранимыми процедурами

Выше в данной главе рассматривались многие темы, касающиеся способов создания хранимой процедуры, а данный раздел в основном посвящен описанию областей применения хранимых процедур. Некоторые из этих областей применения главным образом не требуют пояснений, тогда как другие могут оказаться не столь очевидными для тех, кто впервые осваивается в мире реляционных СУБД. Основные преимущества хранимых процедур перечислены ниже.

- Реализация возможности запуска процессов, которые требуют выполнения процедурных действий.
- Обеспечение защиты.
- Повышение производительности.

Создание вызываемых процессов

Как уже было сказано, хранимая процедура — это своего рода сценарий, хранящийся в базе данных. Иначе говоря, хранимая процедура представляет собой объект базы данных, и в этом есть свое преимущество, поскольку для вызова такого объекта на выполнение не требуется вручную загружать его из файла.

Хранимые процедуры могут применяться для вызова других хранимых процедур (по так называемому принципу **вложения**). В версии SQL Server 2005 допускается вложение хранимых процедур на глубину до 32 уровней. Это дает возможность многократно использовать отдельные хранимые процедуры в основном по такому же принципу, как используются подпрограммы в классических процедурных языках программирования. Синтаксис вызова одной хранимой процедуры из другой является точно таким же, как и синтаксис вызова хранимой процедуры из сценария. Рассмотрим пример создания небольшой хранимой процедуры, предназначенной для выполнения тех же функций, что и проверочный сценарий, который использовался в большинстве разделов данной главы:

```
USE Northwind
GO
CREATE PROC spTestInsert
  @MyDate   smalldatetime
AS
DECLARE   @MyIdent   int
DECLARE   @Return    int
EXEC @Return = spInsertDateValidatedOrder
  @CustomerID = 'ALFKI',
  @EmployeeID = 5,
  @OrderDate  = @MyDate,
  @ShipVia    = 3,
  @Freight    = 5.00,
  @OrderID    = @MyIdent OUTPUT
IF @Return = 0
  SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
  FROM Orders
  WHERE OrderID = @MyIdent
ELSE
  PRINT 'Error Returned was ' + CONVERT(varchar, @Return)
```

После создания этой хранимой процедуры остается только вызвать ее на выполнение, указав вначале допустимую дату, а затем недопустимую дату (чтобы проверить функционирование средств обработки ошибок). Вначале предусмотрим вызов процедуры с допустимой датой:

```
DECLARE @Today smalldatetime
SELECT @Today = GETDATE()
EXEC spTestInsert
   @MyDate = @Today
```

Поскольку дата является сегодняшней, то полученный результат соответствует ожиданиям:

```
The Time of Day in Order Date was truncated
(1 row(s) affected)
OrderID     CustomerID  EmployeeID     OrderDate                ShipName
---------   ----------  -----------    -----------------------  --------
11097       ALFKI       5              2000-09-18 00:00:00.000  NULL
(1 row(s) affected)
```

После этого введем недопустимую дату:

```
EXEC spTestInsert '1/1/2004
```

И в этом случае полученные результаты соответствуют ожиданиям, поскольку действительно формируется сообщение об ошибке:

```
Msg 18054, Level 16, State 1, Procedure spInsertDateValidatedOrder, Line 33
Error 60000, severity 1, state 1 was raised, but no message with that error
number was found in sys.messages. If error is larger than 50000, make sure
the user-defined message is added using sp_addmessage.
Error Returned was 60000
```

Обратите внимание на то, что локальные переменные являются именно таковыми — локальными по отношению к каждой хранимой процедуре. В процессе работы может быть создано пять различных копий переменной `@MyDate`, по одной для каждого из пяти различных вызовов хранимой процедуры, и все эти экземпляры переменной будут независимыми друг от друга.

Использование хранимых процедур для обеспечения защиты данных

Многие разработчики не оценивают всех возможностей применения хранимых процедур в качестве инструментальных средств обеспечения защиты. Хранимые процедуры, во многом как и представления, дают возможность возвращать пользователю требуемый ему набор строк, не предоставляя прав доступа к самой основополагающей таблице. Если какой-то пользователь получает право вызывать на выполнение хранимую процедуру, из этого следует, что пользователь приобретает возможность осуществлять любое действие, предусмотренное в хранимой процедуре, при условии, что это действие предпринимается в контексте самой хранимой процедуры. Таким образом, если некоторый пользователь получает право вызывать на выполнение хранимую процедуру, которая возвращает все строки из таблицы `Customers`, но не получает доступ к самой таблице `Customers`, то пользователь так или иначе приобретает возможность получения данных из таблицы `Customers`, при условии, что для этого

он обращается к хранимой процедуре (тогда как попытка получить непосредственный доступ к таблице окончится неудачей).

Но удобнее всего то, что можно предоставить определенным пользователям доступ в целях модификации данных с помощью хранимой процедуры, а затем дать им возможность осуществлять доступ к основополагающей таблице только для чтения. В таком случае пользователи смогут модифицировать данные в таблице только при том условии, что будут обращаться для этого к хранимой процедуре (и в связи с этим, по всей вероятности, должны будут действовать с учетом определенных бизнес-правил). А для получения требуемых данных пользователи будут иметь возможность затем подключаться непосредственно к СУБД SQL Server с использованием программы Excel, Access или какой-то другой, чтобы формировать собственные определяемые пользователем отчеты, не создавая риска "непреднамеренной" модификации данных.

> Предоставление пользователям возможности непосредственно подключаться к производственной базе данных с помощью такой программы, как Access или Excel, позволяет, с одной стороны, раскрыть для них в системе невероятно привлекательные возможности, а с другой — создать предпосылки исключительно серьезных ошибок. Чем больше прав приобретают пользователи, тем больше увеличивается объем потребляемых ресурсов, а также растет продолжительность выполняемых запросов (естественно, что при этом пользователей мало волнует то, какие при этом возникают отрицательные последствия, связанные с работой самой системы).
>
> Если пользователям действительно необходимо предоставить непосредственный доступ к данным, то следует предусмотреть использование репликации (или проводить периодическую переброску накопленных данных путем резервного копирования и восстановления) для создания полностью отдельной копии базы данных, с которой могли бы работать пользователи. Это позволяет гарантировать предотвращение возникновения блокировок строк, избежать опасностей, связанных с запуском запросов, поглощающих все ресурсы системы, а также исключить целый ряд других проблем.

Использование хранимых процедур для повышения производительности

Вообще говоря, с помощью хранимых процедур можно сделать очень многое для повышения производительности системы. Но необходимо учитывать следующее: как и при использовании большинства других средств, нельзя рассчитывать на абсолютные гарантии. В действительности некоторые процессы, порождаемые с помощью хранимых процедур, способны существенно уменьшить производительность системы, если при проектировании хранимой процедуры не учитывались некоторые важные особенности функционирования хранимых процедур.

Чтобы проанализировать, от каких факторов зависит при этом производительность, рассмотрим показанную на рис. 12.1 схему действий, происходящих при создании хранимой процедуры.

Создание хранимой процедуры начинается с выполнения оператора `CREATE PROC`. После вызова этого оператора на выполнение осуществляется синтаксический анализ запроса хранимой процедуры, позволяющий убедиться в том, что код запроса действительно приемлем для выполнения. Единственное возникающее при этом различие по сравнению с непосредственным вызовом сценария на выполнение состоит

в том, что в операторе CREATE PROC может использоваться так называемое **отложенное преобразование имен**. При отложенном преобразовании имен игнорируется тот факт, что некоторые объекты, используемые в хранимой процедуре, еще не существуют. Благодаря этому появляется возможность создавать такие объекты позже.

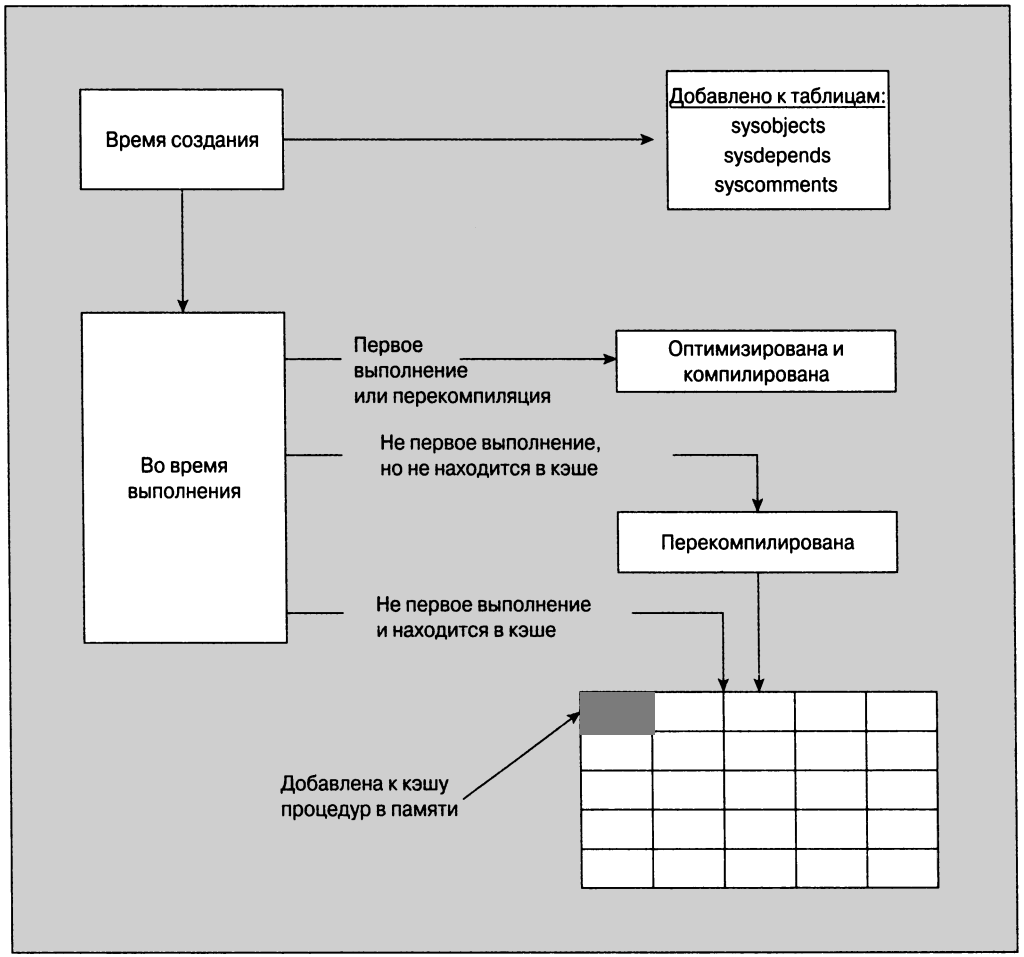


Рис. 12.1. Схема действий, происходящих при создании хранимой процедуры

После создания хранимой процедуры в СУБД не предпринимается больше никаких действий до первого вызова хранимой процедуры на выполнение. А после первого вызова осуществляется оптимизация хранимой процедуры, после чего в системе компилируется и кэшируется план запроса. Этот кэшированный план запроса используется при последующих вызовах хранимой процедуры на выполнение (если с помощью опции WITH RECOMPILE не будет указано, что следует поступить иначе), а не создается каждый раз новый план запроса. Это означает, что при каждом последующем вызове хранимой процедуры удается обойтись без значительной части действий по ее оптимизации и компиляции. Точные показатели достигаемой при этом экономии

зависят от сложности пакета операторов хранимой процедуры, от размеров таблиц, обрабатываемых в этом пакете, и от количества индексов на каждой таблице. Обычно создается впечатление, что экономия ресурсов системы приводит лишь к небольшому сокращению продолжительности выполнения хранимой процедуры (скажем, в большинстве ситуаций такое сокращение может составлять приблизительно одну секунду), но в процентном отношении к исходному времени выполнения такая разница может фактически оказаться весьма значительной (сокращение времени на 1 секунду по сравнению с 2 секундами равносильно ускорению на 100%). Достигнутое ускорение работы может оказаться еще более значительным, если количество вызовов хранимой процедуры велико или если хранимая процедура применяется в цикле.

Предпосылки снижения производительности хранимых процедур

По-видимому, одним из наиболее важных соображений, которые следует учитывать при эксплуатации хранимых процедур, является то, что их оптимизация осуществляется либо перед первым вызовом на выполнение, либо после обновления статистических данных, которые относятся к таблице (таблицам), участвующей в одном из запросов хранимой процедуры (если только в этот процесс не вмешивается администратор базы данных, используя операторы с опцией `WITH RECOMPILE`).

Этот применяемый по умолчанию подход к эксплуатации хранимых процедур, в соответствии с которым оптимизация осуществляется однократно, а вызов на выполнение происходит многократно, способствует повышению производительности приложений, в которых используются хранимые процедуры, но имеет свои недостатки. Если запрос является динамическим (т.е. формируется непосредственно перед вызовом его на выполнение с помощью оператора `EXEC`), то может оказаться, что хранимая процедура была оптимизирована применительно к тем условиям, которые сложились ко времени первого вызова ее на выполнение, притом что в дальнейшем эти условия больше никогда не обнаруживаются; иными словами, для прогона хранимой процедуры используется неправильный план запроса!

Мало того, к указанному сценарию развития событий может привести не только применение динамических запросов в хранимых процедурах. Представьте себе, что в состав приложения входит Web-страница, которая позволяет применять для поиска различные сочетания и комбинации нескольких критериев. Например, предположим, что необходимо ввести в базу данных `Northwind` хранимую процедуру, которая должна поддерживать работу с Web-страницей, позволяющей пользователям выполнять поиск заказов по следующим критериям:

- идентификатор заказчика;
- идентификатор заказа;
- идентификатор товара;
- дата заказа.

Пользователю предоставляется возможность задавать любые сочетания критериев, причем очевидно, что после определения каждого нового критерия область поиска становится более ограниченной, поэтому теоретически определение каждого нового критерия должно способствовать ускорению поиска.

Скорее всего, для решения указанной задачи следует принять такой подход, чтобы было заранее подготовлено несколько запросов и осуществлялся выбор подходящего запроса и вызов его на выполнение в зависимости от того, какие критерии за-

даны пользователем. Таким образом, после первого вызова хранимой процедуры на выполнение в ней обрабатывается ряд операторов IF...ELSE, после чего определяется запрос, подходящий для использования. К сожалению, данный запрос подходит только для данного конкретного вызова хранимой процедуры на выполнение (но неизвестно, насколько часто аналогичная ситуация будет складываться в дальнейшем). После этого первого прогона при каждом вызове хранимой процедуры на выполнение будет вызываться другой запрос, подходящий с точки зрения выбранных на сей раз критериев, но по-прежнему будет использоваться план запроса, сформированный перед первым прогоном хранимой процедуры. Иными словами, производительность запроса действительно будет неоптимальной.

Использование опции WITH RECOMPILE

Эксплуатация хранимых процедур может быть организована таким образом, чтобы применялись их преимущества, связанные с защитой и разделением кода, и вместе с тем исключались недостатки, обусловленные использованием заранее откомпилированного кода. Благодаря этому устраняется та проблема, что при выполнении хранимой процедуры не используется правильный план запроса, поскольку перед каждым очередным прогоном хранимой процедуры создается новый план запроса. Для этого используется опция WITH RECOMPILE, которую можно включить в состав кода двумя способами.

Прежде всего можно задать опцию WITH RECOMPILE непосредственно перед вызовом хранимой процедуры на выполнение, указав ее в операторе вызова, как в следующем примере:

```
EXEC spTestInsert '1/1/2004'
  WITH RECOMPILE
```

В этом операторе СУБД SQL Server передаются указания на то, что необходимо отбросить существующий план выполнения и создать новый, но только на этот раз. Таким образом, при указанном способе применения опции WITH RECOMPILE прогон хранимой процедуры с новым планом выполнения осуществляется только один раз.

Кроме того, требование о неизменной перекомпиляции хранимой процедуры можно задать на постоянной основе, включив опцию WITH RECOMPILE непосредственно в код хранимой процедуры. При подобном подходе опция WITH RECOMPILE задается непосредственно перед ключевым словом AS в операторе CREATE PROC или ALTER PROC.

Если хранимая процедура создана с этой опцией, то ее перекомпиляция происходит при каждом вызове на выполнение, независимо от того, какие опции будут выбраны на этапе прогона.

Расширенные хранимые процедуры

После того как СУБД SQL Server была встроена в инфраструктуру .NET, область применения расширенных хранимых процедур существенно изменилась. В свое время без расширенных хранимых процедур невозможно было обойтись при создании сценариев, наиболее тесно связанных с внутренним функционированием базы данных и применяемых в таких ситуациях, когда простые конструкции T-SQL и другие обычные средства СУБД SQL Server не позволяют решить поставленную задачу.

Но с тех пор как СУБД SQL Server вошла в состав инфраструктуры .NET, позволяющей легко решать задачи обеспечения доступа к файлам операционной системы, поддерживать взаимодействие с другими внешними системами или реализовывать сложные алгоритмы, необходимость в использовании расширенных хранимых процедур в значительность степени снизилась. Безусловно, расширенные хранимые процедуры еще сохранили свою крошечную нишу, поскольку иногда предъявляются столь жесткие требования к производительности, что ее может обеспечить только код, работающий непосредственно под управлением СУБД SQL Server, но в эпоху повсеместного распространения инфраструктуры .NET даже этот способ повышения производительности кажется слишком радикальным.

Поэтому, с учетом направленности настоящей книги, автор ограничится замечанием, что в СУБД SQL Server все еще предоставляется возможность использовать код, вызываемый извне, который оформлен в составе библиотек .DLL, подключаемых к программному обеспечению SQL Server. Для создания расширенных хранимых процедур используются определенные методы программирования на языке низкого уровня. В настоящее время единственными активно поддерживаемыми языками являются C и C++.

Краткие сведения об использовании рекурсии

Рекурсия принадлежит к числу тех методов организации работы, которые используются в программировании не очень часто. Тем не менее она относится к приемам такого типа, что если они действительно требуются, то создается впечатление, будто нельзя найти что-либо иное, позволяющее достичь той же цели. Поэтому автор решил на всякий случай привести краткий обзор тех ситуаций, в которых может потребоваться применение рекурсии.

Краткое определение понятия **рекурсии** состоит в том, что это — метод вызова из кода самого этого кода. Анализ данного определения показывает, какая опасность возникает при использовании рекурсивного кода, т.е. кода, который вызывает сам себя, — если такой вызов произошел один раз, то что может предотвратить осуществление этого вызова снова и снова, до бесконечности? Ответ заключается в том, что все зависит от программиста. Иными словами, необходимо позаботиться о том, чтобы в рекурсивном вызываемом коде была предусмотрена **проверка условия рекурсии** для обеспечения выхода из цепочки рекурсивных вызовов после того, как в этом возникнет необходимость.

К сожалению, трудно найти абсолютно оригинальный пример применения рекурсии, поэтому рассмотрим классический рекурсивный алгоритм, встречающийся почти в любом учебнике, в котором речь идет о рекурсии. Его преимуществом является то, что применяемый алгоритм рекурсии очень прост.

Итак, в данном примере рассматриваются факториалы. Факториалом целого числа называется произведение всех чисел, начиная с единицы и заканчивая самим рассматриваемым числом. Например, факториал числа 5, который обозначается как 5!, равен 120, — т.е. 5*4*3*2*1.

Рассмотрим следующую реализацию алгоритма вычисления факториала в виде рекурсивной хранимой процедуры:

```
CREATE PROC spFactorial
@ValueIn int,
@ValueOut int OUTPUT
```

```
AS
DECLARE @InWorking int
DECLARE @OutWorking int
IF @ValueIn != 1
BEGIN
     SELECT @InWorking = @ValueIn - 1
     EXEC spFactorial @InWorking, @OutWorking OUTPUT
     SELECT @ValueOut = @ValueIn * @OutWorking
END
ELSE
BEGIN
     SELECT @ValueOut = 1
END
RETURN
GO
```

После вызова на выполнение приведенного выше сценария CREATE обнаруживается следующее информационное сообщение:

```
Cannot add rows to sysdepends for the current stored procedure because it
depends on the missing object spFactorial. The stored procedure will still
be created.
```

При создании любых объектов базы данных СУБД SQL Server сохраняет информацию о зависимостях, которая используется при определении того, какие объекты зависят от других. В данном случае применяемая хранимая процедура зависит от самой себя, но СУБД SQL Server не может выявить информацию о зависимости, касающуюся хранимой процедуры, которая еще не существует. И в этом случае возникает дилемма: "Что появилось на свете раньше — курица или яйцо". Но данное сообщение в основном следует рассматривать как информационное, поэтому не нужно предполагать, что оно влечет за собой какие-либо отрицательные последствия.

Итак, хранимая процедура должна принимать в качестве параметра одно целочисленное значение (число, факториал которого должен быть вычислен) и возвращать другое целочисленное значение (значение вычисленного факториала). Изучение этой процедуры вначале вызывает удивление, поскольку в ней, на первый взгляд, нет ничего, что позволило бы вычислить значение факториала за один шаг. Вместо этого в ней просто берется один сомножитель произведения, из которого составляется значение факториала, к нему применяется одна операция, после чего хранимая процедура вызывает сама себя. В следующем вызове снова обрабатывается только одно числовое значение и хранимая процедура снова вызывает сама себя. Такая цепочка вызовов может продолжаться вплоть до достижения глубины рекурсии, равной 32 уровням рекурсии. Но после того как СУБД SQL Server достигает глубины в 32 уровня, активизируется ошибка и обработка прекращается.

Следует учитывать, что любые вызовы сборок .NET засчитываются как дополнительные уровни при подсчете глубины рекурсии, но при проверке достижения глубины рекурсии любые действия, выполняемые в этих сборках, не учитываются.

Проверим приведенную выше рекурсивную хранимую процедуру с помощью небольшого сценария:

```
DECLARE @WorkingOut int
DECLARE @WorkingIn int
SELECT @WorkingIn = 5
EXEC spFactorial @WorkingIn, @WorkingOut OUTPUT

PRINT CAST(@WorkingIn AS varchar) + ' factorial is '
 + CAST(@WorkingOut AS varchar)
```

После вызова этого сценария на выполнения будет получен ожидаемый результат, равный 120:

```
5 factorial is 120
```

При попытке применить другие значения параметра `@WorkingIn` обнаруживается, что хранимая процедура действует в полном соответствии с ожиданиями, если не считать двух довольно существенных описанных ниже возможных нарушений в работе.

- Арифметическое переполнение, возникающее после того, как величина факториала становится слишком большой для данных типа `int` (или даже `bigint`).
- После достижения предельной глубины рекурсии, равной 32 уровням, вызов хранимой процедуры оканчивается аварийно.

Проверить, как происходит арифметическое переполнение, можно очень легко — достаточно задать для вычисления факториала любое большое число (в данном примере подходит любое число больше 13).

Для проверки того, что происходит при достижении предела рекурсии, равного 32 уровням, необходимо внести в хранимую процедуру небольшие изменения. В данном случае мы определим сумму ряда натуральных чисел от единицы до рассматриваемого числа. Применяемые при этом действия весьма напоминают вычисление факториала, за исключением того, что используется сложение, а не умножение. Поэтому сумма чисел от 1 до 5 составляет всего лишь 15, т.е. 5+4+3+2+1, и это означает, что переполнение не возникнет. Создадим новую хранимую процедуру для реализации данного алгоритма; эта процедура почти полностью совпадает с хранимой процедурой вычисления факториала, за исключением нескольких небольших изменений:

```
CREATE PROC spTriangular
@ValueIn int,
@ValueOut int OUTPUT
AS
DECLARE @InWorking int
DECLARE @OutWorking int
IF @ValueIn != 1
BEGIN
       SELECT @InWorking = @ValueIn - 1

       EXEC spTriangular @InWorking, @OutWorking OUTPUT

       SELECT @ValueOut = @ValueIn + @OutWorking
END
ELSE
```

```
BEGIN
      SELECT @ValueOut = 1
END
RETURN
GO
```

Вполне очевидно, что изменения весьма невелики. Аналогичным образом, для проверки этой хранимой процедуры достаточно внести небольшие изменения в оператор вызова хранимой процедуры и в текст оператора PRINT проверочного сценария:

```
DECLARE @WorkingOut int
DECLARE @WorkingIn int
SELECT @WorkingIn = 5
EXEC spTriangular @WorkingIn, @WorkingOut OUTPUT

PRINT CAST(@WorkingIn AS varchar) + ' Triangular is ' + CAST(@WorkingOut AS
varchar)
```

После вызова этого сценария на выполнение со значением @ValueIn, равным 5, будет получено ожидаемое значение 15:

```
5 Triangular is 15
```

Но попытка вызвать эту процедуру на выполнение со значением @ValueIn, большим 32, приводит к возникновению ошибки:

```
Msg 217, Level 16, State 1, Procedure spTriangular, Line 12
Maximum stored procedure, function, trigger, or view nesting level exceeded
(limit 32).
```

К сожалению, какого-либо удобного способа преодоления этого ограничения не существует, поэтому вряд ли что-либо удастся сделать, если нет возможности каким-то образом сегментировать рекурсивные вызовы (осуществлять их прогон на глубину 32 уровня, затем полностью осуществлять выход из стека вызовов, после чего снова вызывать хранимую процедуру на выполнение для обработки очередной порции данных). Но следует учитывать, что большинство рекурсивных функций может быть преобразовано для использования стандартных конструкций организации цикла, а эти конструкции не налагают каких-либо пределов на количество итераций. Поэтому, прежде чем прибегнуть к использованию рекурсии, убедитесь в том, что стоящую перед вами задачу невозможно решить с помощью итерации.

Отладка

Практически применимые средства отладки для СУБД SQL Server впервые появились в версии SQL Server 2000. В указанной версии, во многом как и в версии SQL Server 2005, приходится добиваться почти идеальной настройки параметров и успешного согласования работы нескольких приложений, чтобы обеспечить функционирование средств отладки, но после этого достигаются просто превосходные результаты.

Средства отладки в версии SQL Server 2005 действуют в тесной интеграции с программой Visual Studio и действительно позволяют достигать очень хороших результатов.

Не будет преувеличением сказать, что задача заставить нормально работать средства отладки является очень трудоемкой (а в самых худших случаях даже неосуществимой). В последние годы значительные усилия направлены на обеспечение защиты информации. В связи с этим столь большое число компонентов сервера закрыто для внешних вызовов, что особенно затруднительной стала дистанционная отладка (а именно такая организация работы должна быть обеспечена, если над проектом работает не один разработчик, а несколько). Автор может лишь порекомендовать не отказываться от борьбы и продолжать попытки наладить работу средств отладки; после того как эти средства начнут нормально функционировать, все ваши усилия оправдаются.

Настройка параметров СУБД SQL Server для применения отладки

Некоторые варианты инсталляции могут не требовать каких-либо дополнительных действий для обеспечения работы средств отладки. Тем не менее, если во время инсталляции выбран путь к каталогу, применяемый по умолчанию, и предусмотрена эксплуатация программного обеспечения SQL Server с использованием учетной записи `LocalSystem`, то может оказаться, что средства отладки вообще не работают. Следствием этого становится реальная необходимость выполнить настройку конфигурации, требуемой для отладки службы SQL Server, чтобы она эксплуатировалась с использованием действительной учетной записи пользователя, а именно, записи, предоставляющей доступ с правами администратора к операционной системе, под управлением которой работает СУБД SQL Server.

Безусловно, эксплуатация одной из служб SQL Server с использованием учетной записи, которой предоставлены права администратора, не рекомендуется большинством экспертов по защите данных. Дело в том, что такой способ организации работы приводит к появлению существенной бреши в защите по той причине, что с правами администратора приходится также осуществлять многие другие действия. Достаточно представить себе, что любой пользователь, которому предоставляется право создавать в системе сборки .NET, получает также возможность удалять и перемещать на компьютере любые файлы и даже проводить еще более опасные операции. Такое положение дел допустимо только в системе, применяемой исключительно для разработки. Кроме того, при этом приходится учитывать, что должна использоваться учетная запись администратора компьютера, а не администратора домена.

Запуск программы Debugger

Принципы использования программы Debugger напоминают предусмотренные в отладчиках программ на языке VB или C++; по-видимому, функционирование большинства современных отладчиков организовано по такому же принципу.

Прежде чем перейти к более подробному изложению этой темы, автор обязан предупредить, что программа Debugger, превосходная во многих отношениях, оставляет желать лучшего с точки зрения удобства ее вызова. В последней версии SQL Server эта программа даже больше не встроена в инструментарий обработки запросов, поэтому внимательно следите за описанием того, как осуществляется вызов программы, приведенным в настоящем разделе.

Чтобы перейти к использованию программы Debugger, необходимо запустить программу Visual Studio и создать проект любого типа, который по умолчанию включает источники данных (примером проекта такого типа, который входит в состав

инсталляции SQL Server и включает источники данных в виде узла проекта, является `Integration Services`). Перейдите в окно Server Explorer (команду вызова этого приложения можно найти в меню View), щелкните правой кнопкой мыши на элементе Data Sources, а затем выберите команду New Data Source (если это еще не сделано). Введите информацию о соединении в диалоговом окне Add Connection, как показано на рис. 12.2.

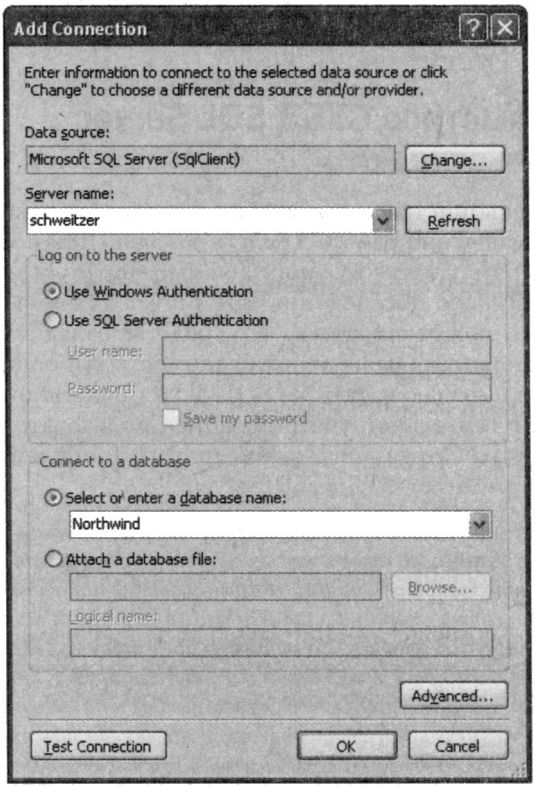


Рис. 12.2. Ввод информации о соединении в диалоговом окне Add Connection

Затем необходимо перейти к обозначению хранимой процедуры (или пользовательской функции), для которой требуется выполнить отладку, и щелкнуть на этом обозначении правой кнопкой мыши. В данном случае перейдите к обозначению хранимой процедуры `spTriangular`, которая рассматривалась в предыдущем разделе, и щелкните правой кнопкой мыши на этом обозначении, после этого выберите команду Step Into Stored Procedure (рис. 12.3).

В результате этого откроется диалоговое окно Run Stored Procedure, в котором имеются приглашения к вводу информации, применяемой в качестве параметров хранимой процедуры (рис. 12.4).

Прежде чем появится возможность вызвать на выполнение хранимую процедуру, должны быть заданы значения всех обязательных параметров (рис. 12.5). В качестве значения `@ValueIn` следует ввести `3`. Это позволяет значительно ограничить количество выполняемых этапов рекурсии и вместе с тем полностью ознакомиться с не-

которыми дополнительными средствами. Для определения значения `@ValueOut` воспользуемся опцией `Set to null`. После этого щелкните на кнопке OK. Полученные результаты показаны на рис. 12.6.

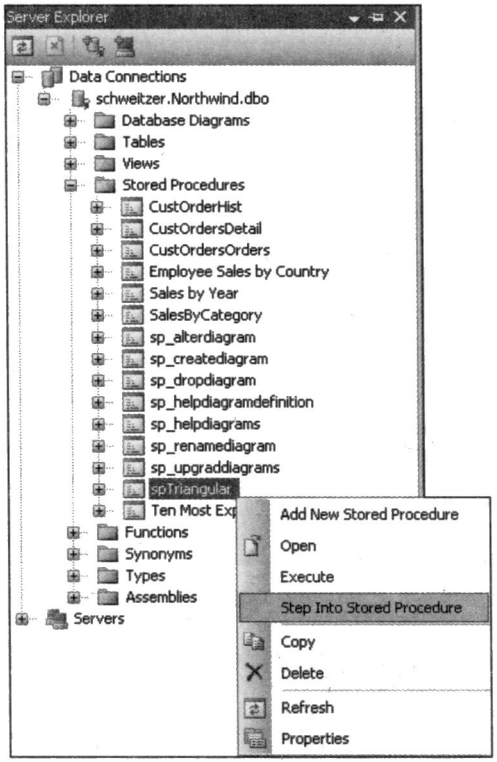


Рис. 12.3. Вызов команды *Step Into Stored Procedure*

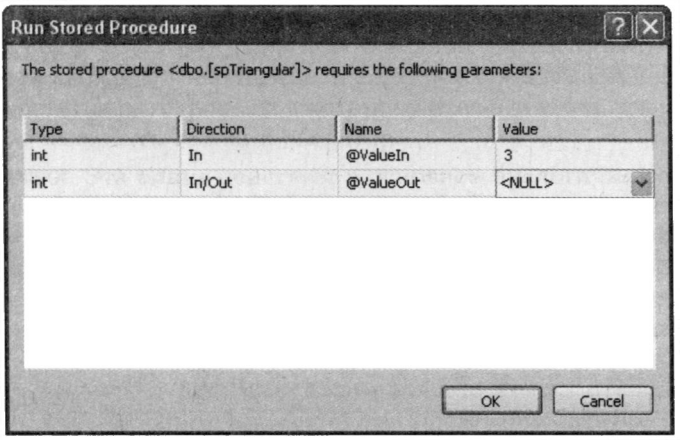


Рис. 12.4. Диалоговое окно *Run Stored Procedure*

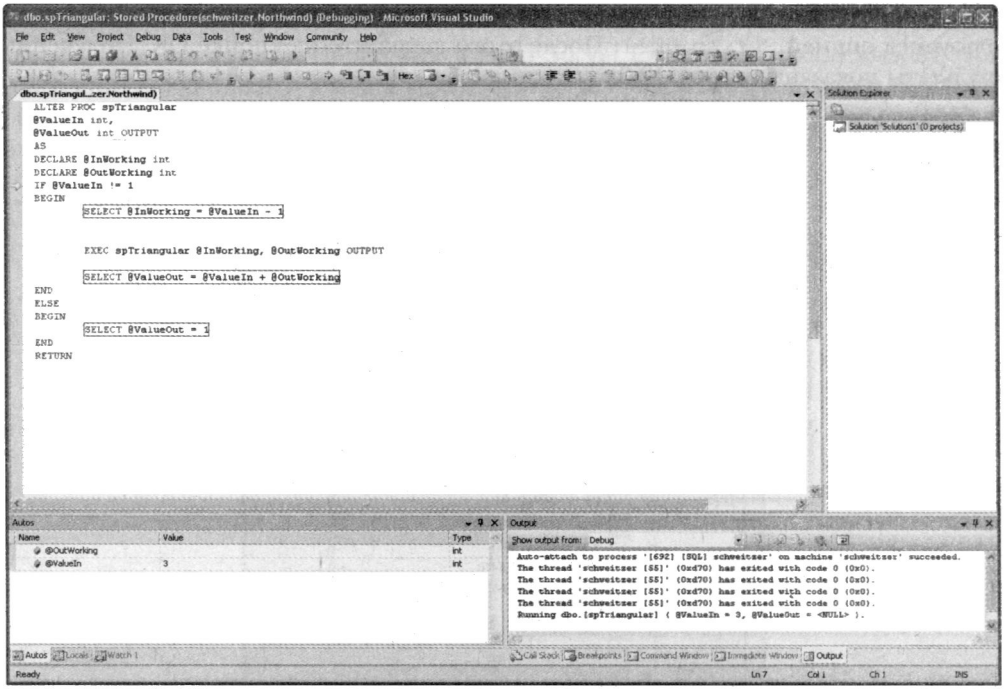


Рис. 12.5. Подготовка к отладке хранимой процедуры `spTriangular`

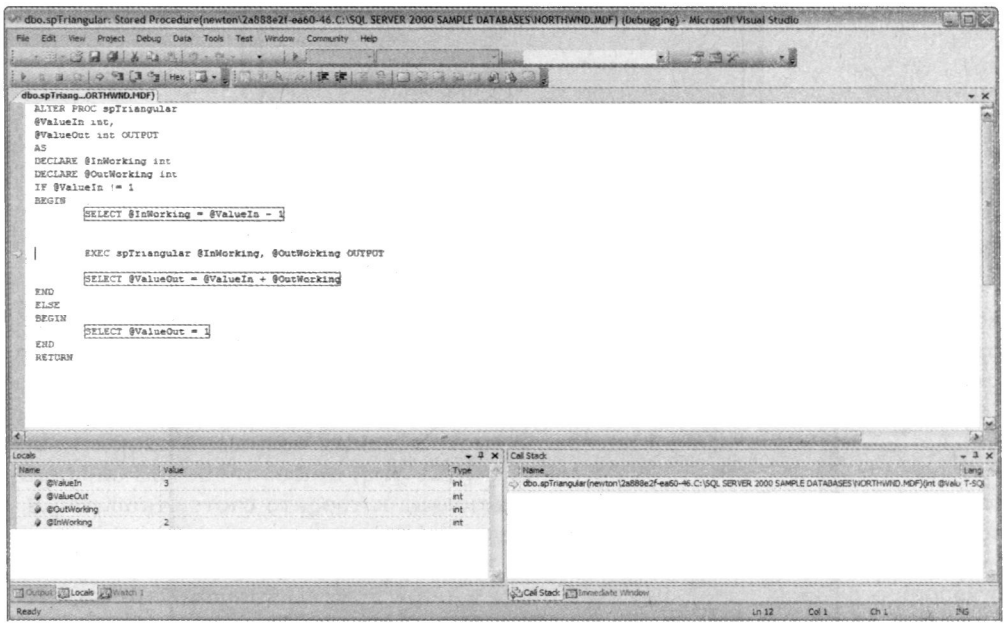


Рис. 12.6. Результаты подготовки хранимой процедуры `spTriangular` *к отладке в программе Debugger*

Компоненты программы Debugger

После того как впервые откроется окно Debugger, необходимо обратить внимание на несколько отличительных особенностей этого окна, описанных ниже.

- Желтая стрелка слева указывает на **текущую строку**. Это — очередная строка кода, которая будет выполнена после выдачи команды "go" или после того, как начнется пошаговое прохождение по операторам кода.
- В верхней части окна имеются пиктограммы, которые обозначают различные опции, в том числе описанные ниже.
- Continue. С помощью этой опции осуществляется прогон до конца хранимой процедуры. После щелчка на соответствующей пиктограмме выполнение хранимой процедуры может быть прекращено преждевременно, только если возникнет ошибка этапа прогона или будет нажата клавиша прерывания.
- Step Into. Эта опция обеспечивает выполнение следующей строки кода и останов, который осуществляется до того, как начнется выполнение следующей строки кода, независимо от того, к какой процедуре или функции относится этот код. Если в строке выполняемого кода имеется вызов хранимой процедуры или функции, то применение опции Step Into равносильно вызову соответствующей хранимой процедуры или функции, внесению информации в стек вызовов, переходу в окно Locals, представляющее вновь вызванную вложенную хранимую процедуру, а не родительскую процедуру, а затем останов на первой строке кода вложенной хранимой процедуры.
- Step Over. Эта опция обеспечивает выполнение всех строк кода, необходимых для достижения следующего оператора, находящегося на том же уровне в стеке вызовов. Если в текущем операторе не вызывается другая хранимая процедура или пользовательская функция, то данная опция действует полностью аналогично опции Step Into. Если же в текущем операторе вызывается другая хранимая процедура или пользовательская функция, то опция Step Over обеспечивает переход к оператору, немедленно следующему за тем оператором, в котором хранимая процедура или пользовательская функция возвращает свое значение.
- Step Out. Эта опция обеспечивает выполнение всех строк кода вплоть до следующей строки кода, относящейся к очередной наивысшей точке в стеке вызовов. Иными словами, выполнение операторов продолжается до тех пор, пока не будет достигнут тот уровень, к которому относится код, вызвавший код того уровня, где находится текущий оператор.
- Run To Cursor. Эта опция действует в основном аналогично сочетанию точки прерывания и оператора Go. Если выбрана эта опция, начинается выполнение кода и продолжается до достижения того места программы, в котором определен текущий курсор. Исключения из этого правила могут наблюдаться, только если обнаруживается точка прерывания, предшествующая местонахождению курсора (в таком случае выполнение операторов прекращается не перед текущим курсором, а в точке прерывания), или достигается конец хранимой процедуры, прежде чем будет вызвана на выполнение строка с определением курсора (как и в таких обстоятельствах, когда определение курсора задано в строке. которая уже встречалась или входит в состав оператора управления процессом выполнения, который еще не был вызван на выполнение).

- Restart. Эта опция осуществляет именно то действие, которое предусмотрено ее названием, — перезапуск. После щелчка на соответствующей пиктограмме восстанавливаются первоначальные значения параметров, инициализируются значения всех переменных и очищается стек вызовов, после чего вся работа возобновляется.
- Stop Debugging. Эта опция также выполняет действие, предусмотренное ее названием, — немедленное прекращение выполнения. Но окно отладчика остается открытым.
- Toggle Breakpoints и Remove All Breakpoints. Кроме всего прочего, задавать точки прерывания можно, щелкнув на левом поле окна с отлаживаемым кодом. Точками прерывания называются такие места программы, где СУБД SQL Server получает команду остановиться при выполнении кода в режиме отладки. Точки прерывания удобно применять при отладке хранимых процедур или функций с большим объемом кода, когда нет необходимости отдельно рассматривать каждую строку и достаточно просто выполнять прогон кода до указанной точки, останавливаясь каждый раз, когда программа дойдет до определенного места.

Кроме того, в программе Debugger имеется несколько окон, которые в дальнейшем изложении будут именоваться окнами контроля состояния; ниже описаны наиболее важные из этих окон.

Окно Locals

Как уже было сказано в начале данной книги, автор во многом рассчитывает на то, что читатель имеет опыт работы с каким-то процедурным языком программирования. Поэтому окно Locals, по-видимому, не будет для вас полностью незнакомым. Кратко объяснить назначение этого окна можно так, что в нем отображаются текущие значения всех переменных, которые в настоящее время находятся в области определения. В ходе пошагового выполнения программы, когда осуществляется вызов вложенных хранимых процедур и возврат управления из этих процедур, перечень переменных в окне Locals может изменяться (наряду с их значениями). Но следует помнить, что отображаются только те переменные, которые заданы в области определения к тому времени, как будет вызван на выполнение следующий оператор.

Для описания каждой переменной или каждого параметра предусмотрены следующие три фрагмента информации:

- имя;
- текущее значение;
- тип данных.

Безусловно, окно Locals является очень удобным, но наилучшим его свойством оказывается то, что оно позволяет редактировать значения любой переменной. Благодаря этому существенно упрощается задача внесения изменений непосредственно в ходе выполнения для проверки того, как осуществляются в хранимой процедуре те или иные действия.

Окно Watch

Окно Watch выполняет в программе Debugger в основном такое же назначение, как в любом современном отладчике. В этом окне могут быть заданы имена переменных, значения которых должны отслеживаться, и даже может быть предусмотрена

активизация точек прерывания с учетом определенных изменений значений контролируемых переменных.

Окно Call Stack

В окне Call Stack ведется перечень всех хранимых процедур и функций, которые в настоящее время активизированы в рассматриваемом процессе. Удобная особенность этого окна состоит в том, что с его помощью можно определять, насколько глубоким является уровень вложенности вызовов процедур, а также переходить с одного уровня вложенности на другой для проверки того, какие значения имеет рассматриваемая в данный момент переменная на каждом уровне.

Окно Output

Окно Output выполняет в основном те функции, которые вытекают из его названия, т.е. применяется для вывода результатов, сформированных в СУБД SQL Server. К ним относятся не только результирующие наборы, но и возвращаемое значение, полученные после завершения выполнения хранимых процедур.

Действия, выполняемые в программе Debugger сразу после ее запуска

На этом предварительное описание работы программы Debugger закончено, поэтому сразу после того, как откроется окно данной программы, мы можем приступить к анализу кода.

Первой исполняемой строкой в рассматриваемой хранимой процедуре является оператор `IF`, поэтому данная строка после запуска программы Debugger становится текущей. Обратите внимание на то, что значения еще не присвоены ни одной переменной, кроме `@ValueIn`, которой присвоено значение параметра вызова хранимой процедуры. Таковым является значение `3`, которое было передано в хранимую процедуру после заполнения соответствующего элемента в диалоговом окне Debug Procedure на предшествующем этапе.

Перейдите на следующую строку, нажав клавишу <F11>, щелкнув на пиктограмме `Step Into` или выбрав соответствующую команду меню.

Фактически значение переменной `@ValueIn` не равно `1`, поэтому на следующем шаге происходит переход к блоку `BEGIN...END`, который определен в операторе `IF`, т.е. выполнение переходит к оператору `SELECT`, в котором инициализируется параметр `@InWorking`. Как будет показано ниже, если бы значение `@ValueIn` действительно было равно единице, то немедленно произошел бы переход к конструкции `ELSE` оператора `IF`.

Перейдите еще на одну строку (нажав клавишу <F11>, щелкнув на пиктограмме Step Into или выбрав команду меню), как показано на рис. 12.7.

Теперь особое внимание следует обратить на то, чему стало равно значение `@InWorking` в окне Locals. Следует отметить, что переменная `@InWorking` приняла правильное значение (значение `@ValueIn` в настоящее время равно `3`, поэтому разность `3 - 1` равна `2`), установленное с помощью оператора `SELECT`. Также не менее важно то, что в окне Call Stack показан лишь текущий экземпляр хранимой процедуры; наличие только одного экземпляра наблюдается в связи с тем, что пошаговый переход во вложенную версию хранимой процедуры еще не произошел.

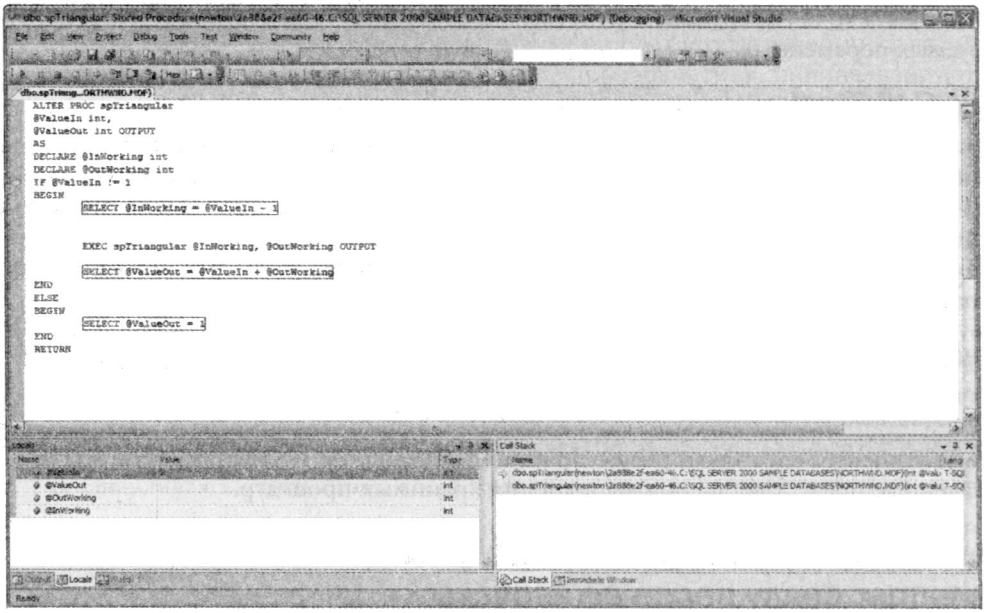


Рис. 12.7. Окна программы Debugger во время пошагового выполнения команд

Продолжим работу и выполним пошаговый переход к следующему оператору. В этом операторе вызывается на выполнение хранимая процедура, поэтому в окне Debugger обнаруживаются различные изменения (рис. 12.8).

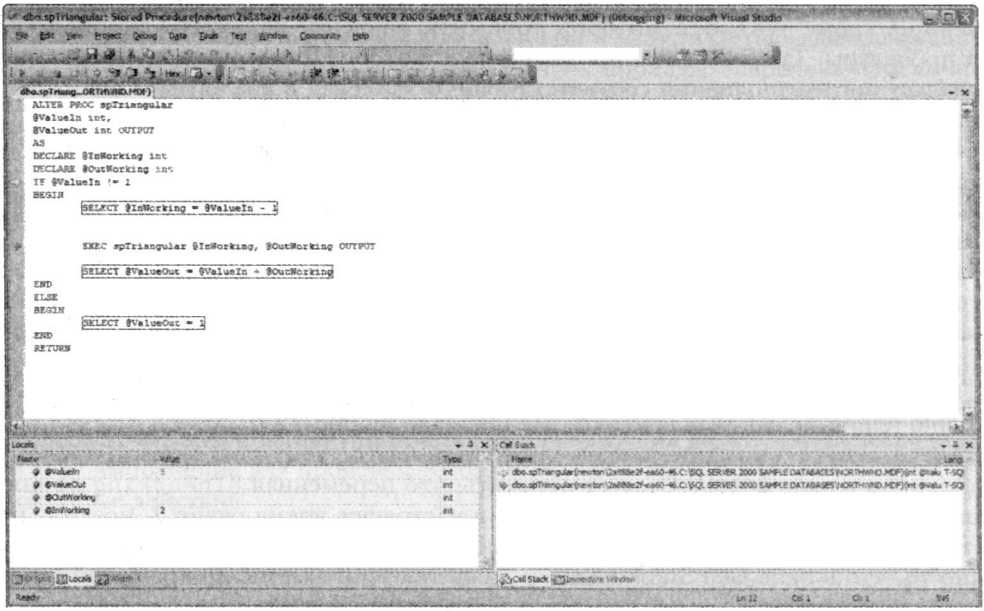


Рис. 12.8. Окна программы Debugger после вызова на выполнение второго экземпляра хранимой процедуры

Важно отметить, что теперь создается впечатление, будто стрелка, которая указывает на текущий оператор, снова была переведена на оператор `IF`. Это связано с тем, что в окне теперь показан новый экземпляр хранимой процедуры. Об этом можно судить по данным, представленным в окне Call Stack, — обратите внимание на то, что в нем теперь показаны два экземпляра хранимой процедуры. В верхней части этого окна синим цветом обозначен текущий экземпляр. Заслуживает также внимания то, что параметр `@ValueIn` имеет значение `2`, — значение, переданное из внешнего экземпляра хранимой процедуры.

Чтобы ознакомиться со значениями переменных в области определения внешнего экземпляра хранимой процедуры, достаточно дважды щелкнуть на строке с обозначением этого экземпляра в окне Call Stack (на строке, отмеченной зеленой стрелкой), после чего обнаружится, что содержимое всех окон снова изменилось (рис. 12.9).

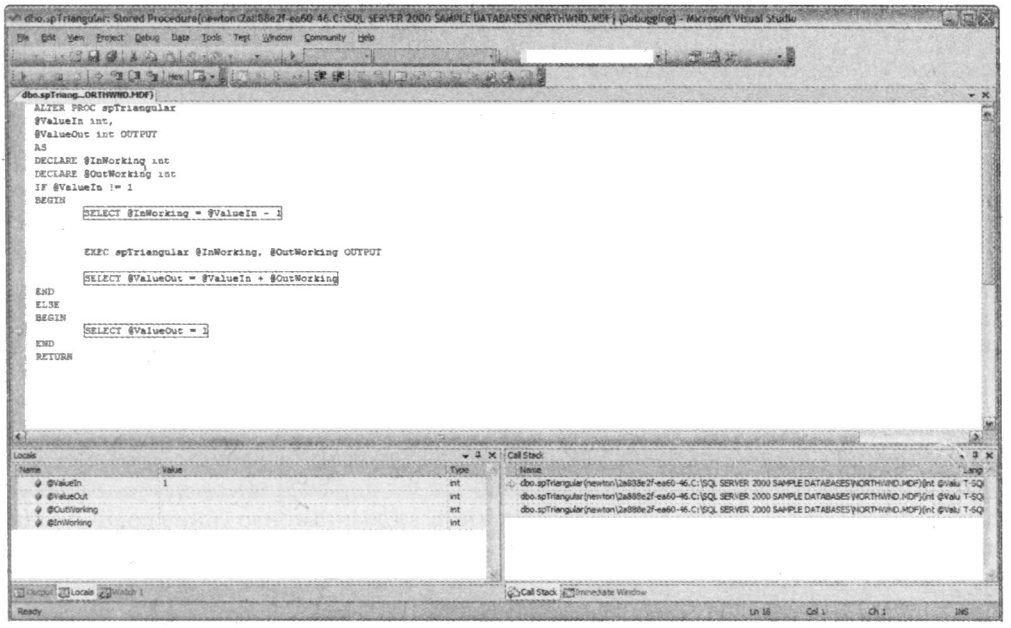


Рис. 12.9. Окна программы Debugger с информацией о первом экземпляре хранимой процедуры

На данном этапе необходимо сделать два замечания. Во-первых, значения переменных вернулись к тем, которые обнаруживались в области действия внешнего (выбранного в настоящее время) экземпляра хранимой процедуры. Во-вторых, предусмотрена другая пиктограмма для обозначения выполняемой в данный момент строки. Эта зеленая стрелка, не применявшаяся ранее, должна обозначать, что указанная строка является текущей в этом экземпляре хранимой процедуры, но ее не следует считать текущей строкой во всем стеке вызовов.

Вернемся к текущему экземпляру, щелкнув на верхнем элементе в окне Call Stack. Затем сделаем еще три пошаговых перехода на следующие операторы. В результате этого должен произойти переход на верхнюю строку (оператор `IF`) в третьем экзем-

пляре хранимой процедуры. Обратите внимание на то, что глубина стека вызовов увеличилась до трех уровней, а значения переменных и параметров в окне Locals снова изменились. Наконец, отметим, что на этот раз параметр `@ValueIn` имеет значение `1`.

После еще одного пошагового перехода в этом коде снова обнаружится небольшое изменение в поведении сценария. На этот раз значение переменной `@ValueIn` действительно равно `1`, поэтому происходит переход в блок `BEGIN...END`, который задан в конструкции `ELSE` (рис. 12.10).

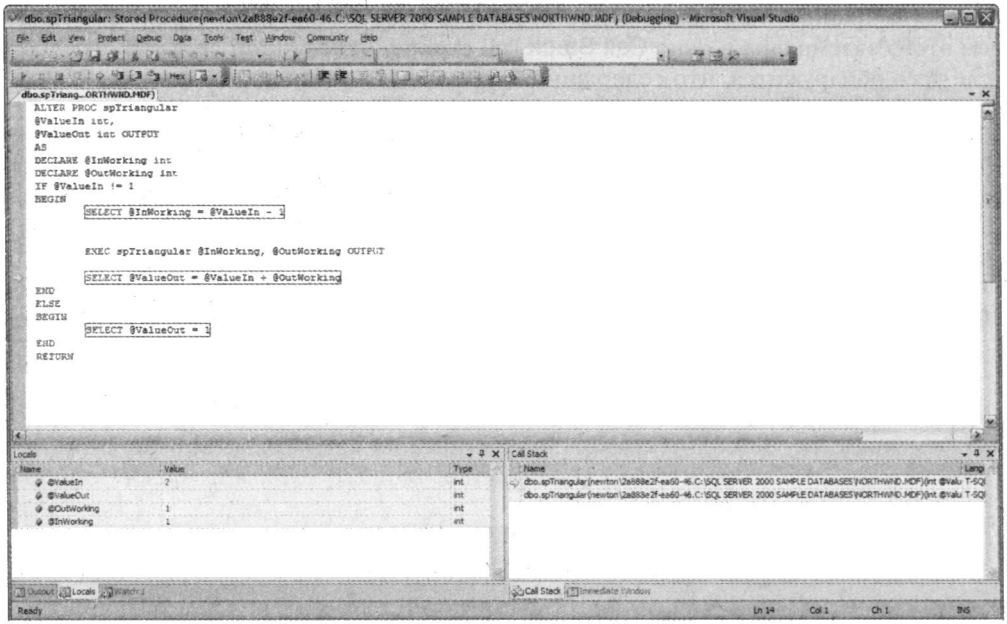


Рис. 12.10. Окна программы Debugger на последнем этапе выполнения рекурсивной процедуры

Итак, достигнут конец стека вызовов, поэтому наступило такое время, когда нужно начать возвращаться назад.

Обратите внимание на то, что стек вызовов снова состоит только из двух уровней. Кроме того, следует отметить, что выходному параметру (`@OutWorking`) присвоено правильное значение.

На этот раз применим немного другой вариант отладки и вызовем на выполнение команду Step Out (с помощью комбинации клавиш <Shift+F11>). После этого невнимательному наблюдателю покажется, будто абсолютно ничего не изменилось.

В данном случае, как гласит старое, затасканное выражение, внешность обманчива. Снова обратите внимание на то, что изменился внешний вид окна Call Stack и появились другие значения в окне Locals. Дело в том, что мы вышли из текущего экземпляра хранимой процедуры и перешли на один уровень выше в стеке вызовов. А если будет продолжено пошаговое выполнение кода (с помощью нажатия клавиши <F11>), то выполнение хранимой процедуры будет закончено, после чего появится заключительная версия окон состояния с соответствующими окончательными значениями. В этой ситуации необходимо учесть важное предостережение! Если вы хотите иметь возможность ознакомиться с теми значениями, которые действительно являются окончательными (такими как выходной параметр с заданным значением), то обязательно воспользуйтесь опцией Step Into для выполнения последней строки программы.

Если же используется опция, с помощью которой выполняется сразу несколько строк кода (такая как Go или Step Out), то появится лишь окно вывода без какой-либо информации об окончательных значениях переменных.

Чтобы исключить возможность последнего варианта развития событий, можно установить точку прерывания в последнем пункте хранимой процедуры, в котором ожидается выполнение оператора `RETURN` для возврата в самый внешний экземпляр хранимой процедуры. Это позволяет использовать любой желаемый режим отладки и вместе с тем обеспечить останов хода выполнения в той точке, где ожидается получение окончательных значений переменных.

Итак, вполне очевидно, что программа Debugger действительно может оказаться весьма удобным средством отладки.

Сборки .NET

Безусловно, тематика, касающаяся сборок .NET, является очень сложной, а возможности, открывающиеся при их использовании для создания чрезвычайно развитых приложений, основанных на применении базы данных, — весьма широкими, но, к сожалению, описание этой темы в основном выходит за рамки рассмотрения настоящей книги.

Сборки .NET — это заранее подготовленные фрагменты кода, которые могут быть связаны с конкретной системой. Сборки предоставляют возможность выполнения чрезвычайно трудоемких и нетривиальных операций. Допустим, какая-то сборка .NET может использоваться в пользовательской функции для получения данных из внешнего источника данных. Предусмотрена даже возможность динамически подключаться к таким источникам данных, как передачи новостей или котировки акций. Следует отметить, что в предыдущих версиях возможность выполнения подобных функций была исключена, поскольку для этого требовались такие средства структурированной обработки данных и обмена данными, которые выходили за рамки существовавших возможностей.

Не будем пока углубляться в слишком подробное описание и рассмотрим лишь синтаксис оператора присоединения сборки к базе данных:

```
CREATE ASSEMBLY  <assembly name>
AUTHORIZATION <owner name>
FROM <path to assembly>
WITH PERMISSION_SET = [SAFE | EXTERNAL_ACCESS | UNSAFE]
```

Та часть приведенного выше оператора, которая состоит из ключевых слов CREATE ASSEMBLY, в основном напоминает по своей структуре большинство рассматривавшихся ранее операторов CREATE: в ней указан тип создаваемого объекта и его имя.

За этим основным определением следует ключевое слово AUTHORIZATION, позволяющая указать контекст, в котором должна всегда эксплуатироваться данная сборка. Иначе говоря, если при использовании сборки требуется доступ к каким-то таблицам, то вместе с ключевым словом AUTHORIZATION должно быть указано имя пользователя (или имя роли), позволяющее определить в системе, должен ли предоставляться доступ к таблицам или нет.

За конструкцией с ключевым словом AUTHORIZATION следует конструкция FROM. В этой конструкции должен быть фактически указан путь к каталогу, в котором находится сборка, а также дано объявление, относящееся к этой сборке.

Наконец, перейдем к рассмотрению конструкции WITH PERMISSION_SET, которая имеет три описанных ниже опции.

- SAFE. Эта опция означает то, что следует из ее имени, — обеспечение защиты. Если задана указанная опция, то доступ к сборке со стороны любого кода, внешнего по отношению к СУБД SQL Server, становится ограниченным. При этом в самой сборке нельзя использовать такие источники данных, как файлы или данные, передаваемые по сети.
- EXTERNAL_ACCESS. Эта опция допускает доступ к таким внешним источникам данных, как файлы или данные, передаваемые по сети, но требует, чтобы сборка все равно эксплуатировалась по такому же принципу, как эксплуатируется управляемый код.
- UNSAFE. Эта опция также обеспечивает действие, которое следует из ее имени, — допускает выполнение небезопасного кода. С ее помощью можно предусмотреть возможность не только обеспечить доступ в сборке к внешним системным объектам, но и эксплуатировать неуправляемый код.

> Риск, которому подвергается система при эксплуатации сборок .NET в любом режиме, отличном от SAFE, настолько велик, что буквально не поддается описанию. Даже в режиме EXTERNAL_ACCESS пользователям системы предоставляется возможность обращаться к сети, файлам или даже к таким внешним источникам данных, которые по существу применяются в режиме доступа с использованием псевдонимов. Это означает, что с помощью внешних объектов могут осуществляться отнюдь не безопасные действия, кроме того, в ходе осуществления такого доступа в сети пользователям может быть присвоен псевдоним любой учетной записи пользователя SQL Server. Поэтому при осуществлении подобного подхода рекомендуем соблюдать исключительную осторожность.

Автор готовит к печати книгу, посвященную описанию средств программирования SQL Server 2005 для профессионалов, с рабочим названием *Professional SQL Server 2005 Programming* (в настоящий момент эта книга находится в производстве в ИД "Вильямс" и выйдет в свет ориентировочно во II квартале 2007 года.), в которой сборки .NET будут описаны более подробно.

Резюме

Безусловно, настоящая глава имеет очень большой объем, но вместе с тем принадлежит к числу наиболее важных глав книги, без освоения тематики которых буквально невозможно заниматься разработкой программного обеспечения для СУБД SQL Server.

Как правило, хранимые процедуры составляют основной объем программного обеспечения, применяемого в СУБД SQL Server. Они позволяют создавать код, который не только может использоваться многократно, но обеспечивает также повышение производительности и расширение функциональных возможностей. Кроме того, в хранимых процедурах используются такие программные конструкции, которые могут оказаться знакомыми для программистов, владеющих другими языками программирования. Тем не менее хранимые процедуры имеют четко определенную сферу применения и не могут заменить в СУБД все прочие виды программного обеспечения.

Преимущества хранимых процедур перечислены ниже.

- Как правило, более высокая производительность.
- Возможность использования в качестве средства защиты от несанкционированного доступа (регламентация доступа к базе данных и обновления данных).
- Многократное использование кода.
- Возможность распределения кода по функциональным модулям (создание специализированных процедур для реализации прикладных алгоритмов).
- Возможность управления ходом выполнения в зависимости от условий, проверяемых на этапе прогона.

Хранимые процедуры характеризуются следующими недостатками.

- Ограниченные возможности переноса кода хранимых процедур с одной платформы на другую (например, в СУБД Oracle применяется совершенно иной способ реализации хранимых процедур).
- При некоторых обстоятельствах неизменное применение одного и того же неправильного плана выполнения (что приводит к существенному снижению производительности).

Очевидно, что хранимые процедуры не следует считать универсальным средством решения всех проблем, но они, вне всякого сомнения, продолжают оставаться основным средством создания программного обеспечения для СУБД SQL Server. В следующей главе будут рассматриваться недавно введенные в версии SQL Server программные конструкции, весьма напоминающие хранимые процедуры, — пользовательские функции.

Упражнения

12.1. Напишите простую хранимую процедуру, которая возвращает требуемую строку с информацией о заказчике из базы данных `Northwind` после получения параметра `CustomerID`.

12.2. Напишите хранимую процедуру, которая принимает данные об идентификаторе территории (`Territory ID`), описании территории (`Territory Description`) и идентификаторе региона (`Region ID`) и вставляет их в виде новой строки в таблицу `Territories` базы данных `Northwind`.

12.3. Модифицируйте хранимую процедуру, созданную при выполнении упр. 12.2, чтобы обеспечить выполнение в ней предварительной проверки наличия внешнего ключа (`RegionID`) перед осуществлением попытки вставки. Если соответствующее значение `RegionID` не существует, активизируйте сообщение об ошибке с текстом "`RegionID is not valid. Please check your RegionID and try again`" (Недопустимое значение `RegionID`. Проверьте значение `RegionID` и повторите ввод).

12.4. Модифицируйте процедуру, созданную при выполнении упр. 12.2, чтобы обеспечить в ней обработку исключительной ситуации, возникшей после обнаружения того, что значение `RegionID` не существует. Организуйте перехват всех прочих ошибок и предусмотрите для них общее сообщение об ошибке: "`An unhandled exception has occurred. Contact your system administrator`" (Возникла необработанная исключительная ситуация. Обратитесь к системному администратору).

13

Пользовательские функции

Пользовательские функции принадлежат к числу наиболее привлекательных объектов SQL Server. Возможность применения пользовательских функций (User Defined Function – UDF) появилась больше пяти лет тому назад, но до сих пор они остаются одними из самых недостаточно используемых и недооцененных объектов SQL Server. Эти объекты произвели потрясающее впечатление на специалистов по базам данных сразу после их внедрения корпорацией Microsoft в версии SQL Server 2000, но со времени появления инфраструктуры .NET пользовательские функции приобрели еще бо́льшие возможности. А с точки зрения читателя настоящей книги, который ознакомился со всеми предыдущими главами, одна из наиболее замечательных особенностей пользовательских функций состоит в том, что ему уже известно почти все, что требуется для создания этих функций. Фактически пользовательские функции чрезвычайно напоминают хранимые процедуры и отличаются от последних только тем, что обладают некоторыми дополнительными характеристиками и возможностями, которые подчеркивают их особенности и обеспечивают применение во многих сложных ситуациях.

В настоящей главе приведено вводное описание пользовательских функций, рассматриваются различные типы пользовательских функций, подчеркивается их отличие от хранимых процедур, а также, безусловно, приводится описание тех ситуаций, в которых может возникнуть необходимость ими воспользоваться. Наконец, даны краткие сведения о том, как можно использовать инфраструктуру .NET для расширения области применения пользовательских функций.

Общее описание пользовательских функций

Пользовательские функции во многом напоминают хранимые процедуры и представляют упорядоченное множество операторов T-SQL, которые заранее оптимизированы, откомпилированы и могут быть вызваны для выполнения работы в виде единого модуля. Основное различие между пользовательскими функциями и хранимыми процедурами состоит в том, как в них осуществляется возврат полученных результатов. А в связи с тем, что для обеспечения предусмотренного в них способа возврата значений в пользовательских функциях должны осуществляться немного другие действия, к их синтаксической структуре предъявляются более жесткие требования по сравнению с хранимыми процедурами.

Для полноты изложения автор обязан подчеркнуть, что между пользовательскими функциями и хранимыми процедурами есть не только сходство, но и различие. Прежде всего, пользовательские функции, безусловно, не могут рассматриваться как замена для хранимых процедур; они представляют собой всего лишь еще один способ организации кода, позволяющий получить дополнительные возможности.

Хранимые процедуры позволяют передавать входные параметры и получать сформированные в них значения в виде возвращаемых выходных параметров. Безусловно, с помощью хранимой процедуры также можно предусмотреть возврат значения в точку вызова, но в действительности это значение предназначено для использования в качестве индикатора успешного или неудачного завершения, а не в качестве возвращаемых данных. Кроме того, хотя с помощью хранимой процедуры можно обеспечить возврат результирующих наборов, фактически эти результирующие наборы нельзя применять для дальнейшей работы с ними в каком-то запросе без предварительной вставки в какую-то таблицу (обычно во временную таблицу).

С другой стороны, при использовании пользовательских функций допускается передавать входные параметры, но выходные параметры в них не предусмотрены. Но отказ от использования выходных параметров компенсируется введением в действие гораздо более надежно формируемого возвращаемого значения. Возвращаемое значение может быть скалярным, как и в случае применения системных переменных, но особенно привлекательным свойством пользовательских функций является то, что тип данных возвращаемого значения не ограничивается только целочисленным типом, как при использовании хранимых процедур. Значения, возвращаемые пользовательской функцией, могут относиться почти к любому типу данных SQL Server (дополнительная информация по этой теме приведена в следующем разделе).

Но возможности формирования возвращаемых значений пользовательской функции не ограничиваются лишь скалярными значениями; допускается также использовать в качестве возвращаемых значений таблицы. Такая возможность является чрезвычайно удобной, и дополнительные сведения по этой теме будут приведены ниже в данной главе.

На этом основании можно отметить, что пользовательские функции подразделяются на два описанных ниже типа.

- Возвращающие скалярное значение.
- Возвращающие таблицу.

Рассмотрим общее определение синтаксиса оператора создания пользовательской функции:

```
CREATE FUNCTION [<schema name>.]<function name>
    ( [ <@parameter name> [AS] [<schema name>.]<scalar data type> [ =
<default value>]
      [ ,...n ] ] )
RETURNS {<scalar type>|TABLE [(<Table Definition>)]}
    [ WITH [ENCRYPTION]|[SCHEMABINDING]|
[ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ] | [EXECUTE AS {
CALLER|SELF|OWNER|<'user name'>} ]
]
[AS] { EXTERNAL NAME <external method> |
BEGIN
    [<function statements>]
    {RETURN <type as defined in RETURNS clause>|RETURN (<SELECT statement>)}
END }[;]
```

Очевидно, что синтаксис оператора `CREATE FUNCTION` является довольно сложным, поскольку возможность применения необязательных частей этой синтаксической структуры зависит от того, какие компоненты были выбраны в других частях оператора создания пользовательской функции. При этом многое зависит от того, возвращает ли функция данные скалярного типа или таблицу, а также создается ли функция, основанная на использовании операторов языка T-SQL, или формируется функциональная структура, в которой применяются средства CLR и .NET. Поэтому ниже различные варианты синтаксической структуры данного оператора рассматриваются отдельно.

Пользовательские функции, возвращающие скалярное значение

По-видимому, пользовательские функции, возвращающие скалярное значение, больше всего напоминают функции, обычно применяемые в программном обеспечении. Как и большинство собственных встроенных функций SQL Server (таких как `GETDATE()` или `USER()`), пользовательские функции такого типа возвращают в вызывающий их сценарий или процедуру скалярное значение.

Как было указано выше, одной из наиболее привлекательных особенностей пользовательских функций является то, что при работе с ними можно не ограничиваться применением в качестве возвращаемых значений данных целочисленного типа, поэтому возвращаемые значения могут относиться к любому допустимому типу данных SQL Server (включая определяемые пользователем типы данных!), кроме данных типа BLOB, курсоров и временных отметок. Способ оформления кода в виде пользовательской функции является весьма привлекательным (даже если необходимо обеспечить лишь возврат целочисленного значения) по двум описанным ниже причинам.

- В хранимых процедурах возвращаемое значение предназначено для использования в качестве индикатора успеха или неудачи, причем в случае неудачного завершения возвращаемое значение предоставляет некоторую конкретную информацию о характере возникшего нарушения в работе, а в пользовательских функциях, напротив, возвращаемое значение служит исключительно в качестве осмысленного фрагмента данных.

- Функции могут вызываться на выполнение как непосредственно встроенные в запрос (например, могут входить в состав оператора SELECT), а хранимые процедуры не предоставляют такой возможности.

Рассмотрим пример создания простой пользовательской функции, который позволяет подчеркнуть такие особенности функций данного типа, благодаря которым они могут использоваться иначе по сравнению с хранимыми процедурами. Безусловно, в качестве иллюстрации можно было бы выбрать пользовательскую функцию, более простую по сравнению с рассматриваемой в данном примере, но она позволяет более наглядно показать различия между хранимыми процедурами и пользовательскими функциями.

По мнению автора, один из наиболее удобных способов использования функции состоит в том, что с ее помощью подготавливаются данные для ввода в поле типа datetime информации о том, что некоторое событие произошло в какой-то определенный день. Обычно при решении такой задачи возникает проблема, связанная с тем, что в поле типа datetime имеется конкретная информация о времени суток, в связи с наличием которой затрудняется сравнение хранимого значения со значением, содержащим только одну дату. Безусловно, с этой проблемой мы уже сталкивались в предыдущих главах, когда требовалось реализовать некоторые операции сравнения значений дат.

Вернемся к базе данных Accounting, которая была создана в одной из предыдущих глав. Предположим, что необходимо получить сведения обо всех заказах, полученных за сегодняшний день. Начнем с того, что внесем в список несколько заказов, в которых проставлена сегодняшняя дата. Для этого просто выберем известные нам идентификаторы заказчиков и служащих из соответствующих таблиц (если в таблицах с данными о заказчиках и служащих базы данных Accounting еще нет строк, то необходимо вставить для обеспечения доступа к ним несколько фиктивных строк). Сам автор для ввода нескольких строк собирается применить небольшой цикл:

```
USE Accounting
DECLARE @Counter  int
SET @Counter = 1
WHILE @Counter <= 10
BEGIN
   INSERT INTO Orders
     VALUES (1, DATEADD(mi,@Counter,GETDATE()), 1)
   SET @Counter = @Counter + 1
END
```

Итак, при выполнении этого сценария происходит вставка десяти строк, в каждой из которых содержится сегодняшняя дата, но строки, следующие друг за другом, отличаются по времени на одну минуту.

Отметим, что при вызове этого сценария непосредственно перед полуночью некоторые из строк могут перескочить на следующие сутки и замысел данного примера не будет раскрыт, поэтому будьте осторожны. Но для всех остальных читателей, кроме полуночников, этот пример будет очень наглядным.

Таким образом, мы можем приступить к выполнению простого сценария, позволяющего определить, какие заказы были введены сегодня. Для этой цели можно попытаться применить примерно такой оператор:

```
SELECT *
FROM Orders
WHERE OrderDate = GETDATE()
```

Но, к сожалению, этот запрос не возвратит ни одной строки. Это связано с тем, что функция GETDATE() возвращает не только дату, но и текущее время, вплоть до миллисекунды. Это означает, что вероятность получения каких-либо данных с помощью запроса, в котором используется функция GETDATE() в чистом виде, является очень низкой, даже если интересующее нас событие произошло в тот же день (чтобы операция сравнения завершилась успешно, должно было быть так, что сравниваемые события произошли в одну и ту же минуту, если используются данные о времени типа smalldatetime, и в течение одной миллисекунды, если используется полный формат datetime).

Обычно при таких обстоятельствах применяется решение, в котором предусматривается прямое и обратное преобразование даты в строку для удаления информации о времени, после чего выполняется операция сравнения.

Соответствующий оператор может выглядеть приблизительно так:

```
SELECT *
FROM Orders
WHERE CONVERT(varchar(12), OrderDate, 101) = CONVERT(varchar(12),
GETDATE(), 101)
```

На сей раз будут получены все строки, в которых столбец OrderDate содержит сегодняшнюю дату, независимо от того, в какое время дня был введен заказ. Но, к сожалению, этот код нельзя назвать наиболее удобным для чтения. К тому же он очень громоздкий. А если в программе приходится предусматривать подобные операции сравнения для целого ряда дат, то соответствующий сценарий приобретает действительно сложный вид.

Поэтому рассмотрим способ выполнения тех же действий, но с помощью простой пользовательской функции. Вначале необходимо создать саму функцию. Эта задача осуществляется с помощью оператора нового типа CREATE FUNCTION, а применяемый при этом синтаксис во многом напоминает синтаксис создания хранимой процедуры. Например, указанную функцию можно реализовать с помощью такого кода:

```
CREATE FUNCTION dbo.DayOnly(@Date datetime)
RETURNS varchar(12)
AS
BEGIN
  RETURN CONVERT(varchar(12), @Date, 101)
END
```

При использовании этой функции дата, возвращаемая функцией GETDATE(), передается в качестве параметра, задача преобразования даты реализуется в теле функции и осуществляется возврат усеченного значения даты.

Чтобы ознакомиться с действием этой функции, внесем соответствующие изменения в приведенный выше запрос:

```
SELECT *
FROM Orders
WHERE dbo.DayOnly(OrderDate) = dbo.DayOnly(GETDATE())
```

После выполнения этого запроса будет получен тот же результирующий набор, как и при использовании обычного запроса. Вполне очевидно, что даже в таком простом

примере, как этот, удобство кода для чтения значительно повышается, а сам вызов осуществляется в основном так же, как и в большинстве языков программирования, которые поддерживают функции. Тем не менее возникает один нюанс — приходится учитывать такое понятие, как схема. По некоторым причинам в СУБД SQL Server поиск объектов, соответствующих именам функции, происходит иначе по сравнению с другими объектами.

На основании приведенного примера можно также сделать вывод, что пользовательские функции предоставляют гораздо больше преимуществ, чем просто повышение удобства чтения. В эти функции можно встраивать запросы, после чего применять функции как метод инкапсуляции для подзапросов. Кроме того, в пользовательских функциях можно инкапсулировать код процедурной реализации почти любых алгоритмов, возвращающий дискретное значение, после чего непосредственно вводить такие функции в запрос.

Рассмотрим очень простой пример подзапроса. Версия этого подзапроса выглядит следующим образом:

```
USE pubs
SELECT Title,
   Price,
   (SELECT AVG(Price) FROM Titles) AS Average, Price - (SELECT AVG(Price)
FROM Titles)
     AS Difference
FROM Titles
WHERE Type='popular_comp'
```

Выполнение приведенного кода приводит к получению довольно простого набора данных:

```
Title                     Price         Average       Difference
------------------------  ------------  ------------  -------------
But Is It User Friendly?  22.9500       14.7662       8.1838
Secrets of Silicon Valley 20.0000       14.7662       5.2338
Net Etiquette             NULL          14.7662       NULL
(3 row(s) affected)
Warning: Null value is eliminated by an aggregate or other SET operation.
```

Предпримем еще одну попытку использования функций, но на этот раз инкапсулируем в виде функций обе операции — и операцию вычисления среднего, и операцию вычитания. Первая функция инкапсулирует операцию вычисления среднего, а вторая — операцию вычитания:

```
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
   RETURN (SELECT AVG(Price) FROM dbo.Titles)
END
GO
CREATE FUNCTION dbo.PriceDifference(@Price money)
RETURNS money
AS
BEGIN
   RETURN @Price - dbo.AveragePrice()
END
```

Обратите внимание на то, что вполне допустимо вкладывать одну пользовательскую функцию в другую.

Следует отметить, что опция WITH SCHEMABINDING осуществляет применительно к функциям такие же действия, как и по отношению к представлениям, – если функция создается с использованием привязки к схеме, то любой объект, от которого зависит функция, не может быть модифицирован или уничтожен без предварительного удаления привязанной к схеме функции. В данном примере фактически привязка к схеме не требовалась, но автор хотел проиллюстрировать использование этой опции, а также подготовить данный пример для применения с той целью, с какой он потребуется немного позже в настоящей главе.

Теперь вызовем тот же запрос на выполнение и применим в нем новую функцию, а не прежнюю модель с подзапросом:

```
USE pubs
SELECT Title,
   Price,
   dbo.AveragePrice() AS Average,
   dbo.PriceDifference(Price) AS Difference
FROM Titles
WHERE Type='popular_comp'
```

В результате формируются те же результаты, что и при использовании подзапроса, но предупреждающее сообщение не появляется!

Следует отметить, что пользовательские функции не только способствуют повышению удобства чтения кода, но и предоставляют дополнительное преимущество, связанное с неоднократным использованием кода. Небольшие примеры, подобные приведенному выше, по-видимому, не позволяют показать, насколько важным является это преимущество, но по мере усложнения применяемых функций экономия трудозатрат разработчиков становится весьма значительной.

Пользовательские функции, которые возвращают таблицу

Возможности применения пользовательских функций в СУБД SQL Server не ограничиваются лишь возвратом с их помощью скалярных значений. Эти функции обеспечивают возврат гораздо более важных объектов – таблиц. Из этого следуют такие возможности, которые трудно представить себе сразу же, но отметим, что возвращаемая таблица в большинстве обстоятельств доступна для применения в основном с использованием таких же способов, как и любая другая таблица. Результаты, возвращаемые функцией, можно включать в состав операндов операции JOIN и даже применять к ним условия конструкций WHERE. Благодаря этому открываются действительно заманчивые перспективы.

Изменения, которые должны быть внесены в пользовательскую функцию для получения возможности использовать таблицу в качестве возвращаемого значения, не являются сложными, поскольку, что касается таких функций, таблица рассматривается наряду с любым другим типом данных SQL Server. Чтобы проиллюстрировать сказанное, вначале создадим относительно простую функцию:

```
USE pubs
GO
```

```
CREATE FUNCTION dbo.fnAuthorList()
RETURNS TABLE
AS
RETURN (SELECT au_id,
          au_lname + ', ' + au_fname AS au_name,
          address AS address1,
          city + ', ' + state + ' ' + zip AS address2
        FROM authors)
GO
```

В этой функции выполняется возврат таблицы, состоящей из строк, полученных с помощью оператора SELECT, а также несложное форматирование: конкатенация фамилии и имени, разделенных запятыми, а также конкатенация трех компонентов адреса для возврата значений столбца address2.

С этого момента созданная функция может использоваться по такому же принципу, как таблица, за единственным исключением — как было указано при описании скалярных функций, при обозначении имени функции необходимо использовать соглашение об именовании, касающееся двухкомпонентных имен:

```
SELECT *
FROM dbo.fnAuthorList()
```

Полученные результаты имеют довольно большой объем (табл. 13.1), поэтому приведены лишь начало и конец таблицы, но эти данные наглядно демонстрируют суть действий, выполняемых данной функцией.

Таблица 13.1. Результаты выборки данных с применением функции fnAuthorList()

Столбец au_id	Столбец au_name	Столбец address1	Столбец address2
172-32-1176	White, Johnson	10932 Bigge Rd.	Menlo Park, CA 94025
213-46-8915	Green, Marjorie	309 63rd St. #411	Oakland, CA 94618
238-95-7766	Carson, Cheryl	539 Darwin Ln.	Berkeley, CA 94705
...	...	...	...
...	...	...	...
893-72-1158	McBadden, Heather	301 Putnam	Vacaville, CA 95688
899-46-2035	Ringer, Anne	67 Seventh Av.	Salt Lake City, UT 84152
998-72-3567	Ringer, Albert	67 Seventh Av.	Salt Lake City, UT 84152

Но приведенный выше пример еще не позволяет судить обо всех возможностях пользовательских функций. Ведь таблицу, сформированную в этом примере, вполне можно было получить столь же просто (а в действительности даже проще) с помощью представления. Но если бы нам потребовалось параметризовать представление или предусмотреть возможность его применения для получения информации только о тех авторах, которые написали книги, распроданные по меньшей мере в определенном количестве, то задание стало бы гораздо сложнее. Безусловно, можно было бы решить указанные задачи, соединив полученную таблицу еще с одной или с двумя таблицами, но отметим, что и в этом случае применяемый код стал бы гораздо более громоздким и могла бы возникнуть необходимость включить в представление дополнительный столбец, без которого в других условиях можно было обойтись (столбец с данными о количестве проданных книг), а затем применить конструкцию WHERE.

В результате реализации указанного подхода был бы получен примерно такой сценарий создания представления:

```
-- Создание представления
CREATE VIEW vSalesCount
AS
   SELECT au.au_id,
      au.au_lname + ', ' + au.au_fname AS au_name,
      au.address,
      au.city + ', ' + au.state + ' ' + zip AS address2,
      SUM(s.qty) As SalesCount
   FROM authors au
   JOIN titleauthor ta
      ON au.au_id = ta.au_id
   JOIN sales s
      ON ta.title_id = s.title_id
   GROUP BY au.au_id,
      au.au_lname + ', ' + au.au_fname,
      au.address,
      au.city + ', ' + au.state + ' ' + zip
GO
```

Разумеется, этот сценарий позволяет решить поставленную задачу, но с учетом некоторых нюансов. Во-первых, невозможно предусмотреть применение параметров непосредственно в самом представлении, поэтому в запрос придется ввести конструкцию WHERE. Во-вторых, потребуется предоставить конкретный список выборки для исключения столбца vSalesCount (напомним, что задача состоит в том, чтобы показать авторов, количество проданных книг которых превысило указанное значение, но не обязательно фактические объемы сбыта книг этих авторов). Поэтому приходится применять для получения данных оператор

```
SELECT au_name, address, Address2 FROM vSalesCount
WHERE SalesCount > 25
```

Выполнение этого оператора приводит к получению результатов, которые выглядят так, как показано в табл. 13.2.

Таблица 13.2. Результаты выполнения запроса, в котором используется представление vSalesCount

Столбец au_name	Столбец address	Столбец address2
Green, Marjorie	309 63rd St. #411	Oakland, CA 94618
Carson, Cheryl	589 Darwin Ln.	Berkeley, CA 94705
O'Leary, Michael	22 Cleveland Av. #14	San Jose, CA 95128
Dull, Ann	3410 Blonde St.	Palo Alto, CA 94301
DeFrance, Michel	3 Balding Pl.	Gary, IN 46403
MacFeather, Stearns	44 Upland Hts.	Oakland, CA 94612
Panteley, Sylvia	1956 Arlington Pl.	Rockville, MD 20853
Hunter, Sheryl	3410 Blonde St.	Palo Alto, CA 94301
Ringer, Anne	67 Seventh Av.	Salt Lake City, UT 84152
Ringer, Albert	67 Seventh Av.	Salt Lake City, UT 84152

Но если вместо этого все необходимые операторы будут инкапсулированы в виде функции, то применяемый способ решения задачи значительно упростится:

```
USE pubs
GO
CREATE FUNCTION dbo.fnSalesCount(@SalesQty bigint)
RETURNS TABLE
AS
RETURN (SELECT au.au_id,
          au.au_lname + ', ' + au.au_fname AS au_name,
          au.address,
          au.city + ', ' + au.state + ' ' + zip AS Address2
        FROM authors au
        JOIN titleauthor ta
          ON au.au_id = ta.au_id
        JOIN sales s
          ON ta.title_id = s.title_id
        GROUP BY au.au_id,
          au.au_lname + ', ' + au.au_fname,
          au.address,
          au.city + ', ' + au.state + ' ' + zip
        HAVING SUM(qty) > @SalesQty
        )
GO
```

Таким образом, мы провели весьма неплохую предварительную подготовку, и, чтобы выполнить нужный нам запрос, достаточно вызвать функцию и передать ей параметр:

```
SELECT *
FROM dbo.fnSalesCount(25)
```

При этом будет получен точно такой же результирующий набор, но для этого не приходится применять конструкцию WHERE, исключать ненужные столбцы с помощью списка выборки, а также испытывать какие-либо другие затруднения. Кроме того, единожды созданную функцию можно вызывать на выполнение снова и снова, не прибегая каждый раз к формированию текста запроса с помощью надоевшего метода "вырезки и вставки". К тому же следует отметить, что, даже несмотря на возможность достижения аналогичных результатов с помощью хранимой процедуры и оператора EXEC, таблицу, полученную с помощью хранимой процедуры, нельзя непосредственно соединить с другой таблицей.

Чтобы проиллюстрировать сказанное, немного дополним рассматриваемый пример. Предположим, что менеджер из отдела сбыта хочет иметь возможность формировать отчет со списком, в котором указан каждый автор и его издатель (издатели), если количество проданных книг этого автора превышает 25. Для получения данного отчета необходимо выполнить операцию соединения, но хранимая процедура не позволяет решить такую задачу непосредственно, поэтому нелегко сразу же найти способ выхода из этого затруднения. (Автор знает, как это сделать с помощью определенного процесса, состоящего из нескольких шагов, но этот процесс является довольно трудоемким.) А в случае использования приведенной выше функции никаких проблем не возникает:

```
SELECT DISTINCT p.pub_name, a.au_name
FROM dbo.fnSalesCount(25) AS a
JOIN titleauthor AS ta
  ON a.au_id = ta.au_id
JOIN titles AS t
  ON ta.title_id = t.title_id
JOIN publishers AS p
  ON t.pub_id = p.pub_id
```

В результате выполнения этого оператора формируется список интересующих нас авторов с указанием всех тех издателей, которые опубликовали их книги (табл. 13.3).

Таблица 13.3. Результаты применения функции `fnSalesCount()` в сложном запросе с операторами соединения

Столбец pub_name	Столбец au_name
Algodata Infosystems	Carson, Cheryl
Binnet & Hardley	DeFrance, Michel
Algodata Infosystems	Dull, Ann
Algodata Infosystems	Green, Marjorie
New Moon Books	Green, Marjorie
Algodata Infosystems	Hunter, Sheryl
Algodata Infosystems	MacFeather, Stearns
Binnet & Hardley	MacFeather, Stearns
Algodata Infosystems	O'Leary, Michael
Binnet & Hardley	O'Leary, Michael
Binnet & Hardley	Panteley, Sylvia
New Moon Books	Ringer, Albert
Binnet & Hardley	Ringer, Anne
New Moon Books	Ringer, Anne

Вполне очевидно, что в приведенном выше операторе функция участвовала в операции соединения таким образом, как если бы она была таблицей или представлением. Единственное видимое различие состоит в том, что после имени функции указаны круглые скобки и задан параметр.

Даже если бы возможности пользовательских функций не выходили за рамки описанных выше, и это было бы просто замечательно, но иногда для решения поставленной задачи невозможно ограничиться лишь единственным оператором SELECT. В некоторых случаях требуются такие функции, которые уже не подлежат замене с помощью параметризованного представления. И действительно, даже на примере некоторых описанных выше скалярных функций в определенных обстоятельствах для получения необходимых результатов может потребоваться выполнить несколько операторов. Пользовательские функции вполне обеспечивают реализацию такого подхода. Безусловно, как показывает приведенный выше пример функции с одним оператором, ничто не препятствует использованию функций для формирования и возврата таблиц, созданных с помощью нескольких операторов. Единственное значительное различие между функциями с одним и несколькими операторами состоит в том, что в последнем случае необходимо присвоить возвращаемой таблице имя и

определить ее метаданные (во многом аналогично тому, как при использовании временных таблиц).

В данном примере приходится сталкиваться с одной из очень распространенных проблем в мире реляционных баз данных — с задачей построения иерархий. Предположим, что отдел кадров компании Northwind обращается к вам с просьбой решить следующую задачу. В базе данных Northwind компании Northwind имеется таблица Employees, на которой задана односторонняя связь, определенная на столбце ReportsTo для каждого служащего и показывающая, кому подчиняется данный служащий. Это означает, что установить связь между подчиненным и его непосредственным руководителем можно, связав идентификатор служащего, хранящийся в столбце ReportsTo, с идентификатором другого служащего, который хранится в столбце EmployeeID. В отделах кадров очень часто возникает необходимость сформировать на основании данных о непосредственной подчиненности служащих иерархическое дерево подчиненности, т.е. представленные в виде организационной схемы списки всех сотрудников, которые подчиняются прямо или косвенно тому или иному руководителю.

На первый взгляд такая задача кажется довольно простой, ведь если требуется составить список всех служащих, которые подчиняются, скажем, служащему Andrew Fuller, то можно написать примерно такой запрос, в котором таблица Employees соединяется сама с собой:

```
Use Northwind
SELECT Emp.EmployeeID, Emp.LastName, Emp.FirstName, Emp.ReportsTo
FROM Employees AS Emp
JOIN Employees AS Mgr
  ON Mgr.EmployeeID = Emp.ReportsTo
WHERE Mgr.LastName = 'Fuller'
  AND Mgr.FirstName = 'Andrew'
```

Опять-таки, на первый взгляд, может показаться, что по условиям задачи должны быть получены примерно такие результаты:

```
EmployeeID       LastName          FirstName         ReportsTo
-------------    -------------     -------------     -------------
1                Davolio           Nancy             2
3                Leverling         Janet             2
4                Peacock           Margaret          2
5                Buchanan          Steven            2
8                Callahan          Laura             2
(5 row(s) affected)
```

Но в действительности на этом задача не исчерпывается. Дело в том, что должна быть получена информация обо всех служащих, которые принадлежат к дереву подчиненности, начинающемуся от служащего Andrew Fuller и включающему не только тех, кто подчиняется служащему Andrew Fuller, но и тех, кто подчиняется служащим, подчиняющимся служащему Andrew Fuller, и т.д. В частности, просмотр всех строк в таблице Employees базы данных Northwind показывает, что многие служащие подчиняются служащему Steven Buchanan, но информация о них отсутствует в результатах данного запроса.

Как показывает преподавательский опыт автора, наиболее сообразительные или опытные студенты тут же заявляют, что в этом нет никаких проблем, поскольку достаточно просто еще раз включить таблицу `Employees` *в операцию соединения. Безусловно, подобное решение было бы осуществимо при наличии весьма небольшого набора данных или в любой другой ситуации, когда количество уровней иерархии ограничено, но такие условия складываются далеко не всегда. Вполне может быть так, что в компании есть служащие, подчиняющиеся служащему* `Steven Buchanan`*, а также другие служащие, подчиняющиеся служащим, которые подчиняются служащему* `Steven Buchanan`*, и т.д., иными словами, глубина дерева подчиненности может увеличиваться практически неограниченно. Поэтому необходимо найти другой подход.*

В действительности требуется функция, которая возвращала бы информацию обо всех уровнях иерархии, расположенных ниже заданного значения идентификатора служащего `EmployeeID` (следовательно, идентификатора служащего, выполняющего роль руководителя, — `ManagerID`). Способ, позволяющий наилучшим образом решить эту задачу, представляет собой классический пример **рекурсии**. Если в каком-то блоке кода вызывается сам этот код, такой вызов рассматривается как рекурсивный. В предыдущей главе уже рассматривались такие примеры рекурсивного кода, как хранимые процедуры `spFactorial` и `spTriangular`. А в данном случае возникает задача, алгоритм решения которой может выглядеть так, как описано ниже.

1. Составить список всех служащих, которые подчиняются интересующему нас служащему, выполняющему роль руководителя.
2. Для каждого служащего, внесенного в список в шаге 1, составить список подчиняющихся ему служащих.
3. Повторять шаг 2 до тех пор, пока не удастся больше найти служащих, подчиняющихся кому-либо из служащих, внесенных в списки в ходе выполнения предыдущих шагов.

Приведенная формулировка представляет собой классический пример рекурсии. Это означает, что для обеспечения работы функции необходимо использовать операторы нескольких типов: одни из них должны обеспечивать определение того, какой уровень должен стать текущим, а другие (по меньшей мере один) должны снова вызывать ту же функцию для перехода на очередной, более низкий уровень иерархии.

Следует учитывать, что на пользовательские функции распространяются те же ограничения на пределы рекурсии, что и на хранимые процедуры. Это означает, что допускается переход не больше чем на 32 уровня рекурсии, поэтому если возникает вероятность достижения указанного предела, то при создании кода приходится применять определенный творческий подход для предотвращения ошибок.

Реализуем описанный замысел рекурсивного алгоритма в виде функции. Следует отметить, что в объявлении этой функции внесено несколько изменений. Дело в том, что на этот раз требуется связать с возвращаемым значением имя переменной (в данном случае `@Reports`), поскольку к нему приходится обращаться каждый раз, когда для выработки результата могут использоваться различные операторы. Кроме того, необходимо объявить возвращаемую таблицу; это позволяет СУБД SQL Server определить, предпринимается ли попытка вставки данных в эту таблицу перед ее возвратом в вызывающую процедуру:

```
CREATE FUNCTION dbo.fnGetReports
      (@EmployeeID AS int)
      RETURNS @Reports TABLE
      (
      EmployeeID     int         NOT NULL,
      ReportsToID    int         NULL
      )
AS
BEGIN
/* Эта функция должна вызываться рекурсивно, по одному разу для каждого служащего,
** играющего роль подчиненного (чтобы убедиться в том, что он не рассматривается
** как подчиненный по отношению к самому себе), поэтому требуется переменная,
** позволяющая следить за тем, данные о каком служащем рассматриваются
** в данный момент */
DECLARE @Employee AS int
/* В следующем операторе производится вставка данных о рассматриваемом
** служащем в рабочую таблицу. Важно отметить, что первая запись должна служить
** чем-то вроде образца для рекурсивной функции, и этот образец создается */
INSERT INTO @Reports
   SELECT EmployeeID, ReportsTo
   FROM Employees
   WHERE EmployeeID = @EmployeeID
/* Теперь производится выборка образца для рекурсивных вызовов. Для этого,
** по-видимому, было бы лучше применить курсор, но речь об этом пойдет в одной
** из следующих глав... */
SELECT @Employee = MIN(EmployeeID)
FROM Employees
WHERE ReportsTo = @EmployeeID
/* Следующие операции также, по-видимому, было бы лучше выполнить с помощью
** курсора, но на данный момент эта тема еще не пройдена, поэтому соответствующие
** действия просто моделируются. Обратите внимание на то, что вызов функции
** является рекурсивным! */
WHILE @Employee IS NOT NULL
   BEGIN
      INSERT INTO @Reports
          SELECT *
          FROM fnGetReports(@Employee)
          SELECT @Employee = MIN(EmployeeID)
          FROM Employees
          WHERE EmployeeID > @Employee
            AND ReportsTo = @EmployeeID
   END
RETURN
END
GO
```

В определении функции, приведенном выше, предусмотрено получение лишь минимальной информации о служащем и его руководителе, поскольку, если бы потребовалось получить дополнительную информацию, можно было бы просто снова выполнить соединение с таблицей Employees. Кроме того, автор позволил себе немного расширить рамки толкования требований, предъявленных в условиях задачи, поэтому включил в результаты и данные о самом указанном руководителе. Это было сделано в основном для упрощения реализации рекурсивного алгоритма, а также для предоставления своего рода исходного результата для результирующего набора.

В качестве иллюстрации рассмотрим пример формируемых результатов — служащий Andrew Fuller имеет идентификатор служащего EmployeeID, равный 2, поэтому непосредственно укажем данный идентификатор в вызове функции:

```
SELECT * FROM fnGetReports(2)
```

В результате выполнения этого запроса будет получена не только ранее выявленная информация о пяти служащих, подчиняющихся служащему Andrew Fuller, но также информация о тех, кто подчиняется служащему Steven Buchanan (который подчиняется мистеру Фуллеру) и о самом мистере Фуллере (напомним, что было решено включать в качестве исходной точки информацию о служащем, находящемся на самой вершине дерева подчиненности):

```
EmployeeID              ReportsToID
-----------------       -----------------
2                       NULL
1                       2
3                       2
4                       2
5                       2
6                       5
7                       5
9                       5
8                       2
(9 row(s) affected)
```

Как оказалось, в полученные результаты вошли сведения обо всех служащих компании Northwind (эти результаты на компьютере читателя могут оказаться другими, если была также введена информация о других служащих), причем проверка вручную полученных данных с помощью информации, представленной в столбце ReportsTo, показывает, что полученные результаты действительно соответствуют ожидаемым. Но чтобы выполнить еще одну проверку, можно вызвать рассматриваемую функцию, указав идентификатор служащего Steven Buchanan (который равен 5):

```
SELECT * FROM fnGetReports(5)
EmployeeID              ReportsToID
---------------         ----------------
5                       2
6                       5
7                       5
9                       5
(4 row(s) affected)
```

Как и следовало ожидать, объем полученных результатов уменьшился. Теперь мы можем перейти к осуществлению завершающего шага и применить операцию соединения к полученным и исходным данным. Для этого воспользуемся почти таким же запросом, с помощью которого была предпринята первая попытка выяснить, кто является подчиненным служащего Andrew Fuller:

```
DECLARE @EmployeeID int
SELECT @EmployeeID = EmployeeID
   FROM Employees
   WHERE LastName = 'Fuller'
   AND FirstName = 'Andrew'
```

```
SELECT Emp.EmployeeID, Emp.LastName, Emp.FirstName, Mgr.LastName AS ReportsTo
FROM Employees AS Emp
JOIN dbo.fnGetReports(@EmployeeID) AS gr
  ON gr.EmployeeID = Emp.EmployeeID
JOIN Employees AS Mgr
  ON Mgr.EmployeeID = gr.ReportsToID
```

В результате будет получена информация обо всех восьми служащих, которые прямо или косвенно подчиняются мистеру Фуллеру:

```
EmployeeID  LastName    FirstName   ReportsTo
----------  ----------  ----------  ----------
1           Davolio     Nancy       Fuller
3           Leverling   Janet       Fuller
4           Peacock     Margaret    Fuller
5           Buchanan    Steven      Fuller
6           Suyama      Michael     Buchanan
7           King        Robert      Buchanan
9           Dodsworth   Anne        Buchanan
8           Callahan    Laura       Fuller
(8 row(s) affected)
```

Любопытно отметить, что в результатах выполнения последнего запроса отсутствует информация о самом мистере Фуллере, хотя, как уже было сказано, информация о нем присутствует в результатах выполнения функции. Причина отсутствия данных о служащем `Andrew Fuller` состоит в том, что в результирующем наборе значение поля `ReportsTo` в строке с данными об этом служащем составляет `NULL`, поэтому отсутствует информация, на основании которой могло бы быть выполнено соединение с таблицей `Employees`. Таким образом, исключение данных об этом служащем происходит на этапе выполнения запроса, а не функции.

Таким образом, приведенный выше пример показывает, что пользовательские функции позволяют использовать для формирования табличных результатов очень сложный код, но поскольку полученные результаты представлены в виде таблицы, то их можно использовать для осуществления дальнейших операций наравне с любой другой таблицей.

Требования по обеспечению детерминированного выполнения функций

Одним из важных требований к пользовательским функциям является обеспечение их детерминированного выполнения. До сих пор в данной книге требования по обеспечению детерминированного выполнения рассматривались применительно к тому, как с помощью СУБД SQL Server осуществляется поиск данных в таблице, на которой задан индекс. В соответствии с этими требованиями индекс должен определять каждый индексируемый им элемент данных детерминированно (т.е. полностью однозначно). А что касается функций, то требования по обеспечению их детерминированного выполнения предъявляются в связи с тем, что некоторые функции предоставляют данные для выполнения операций с индексированными объектами (например, с вычисленными столбцами или индексированными представлениями).

По указанному признаку пользовательские функции подразделяются на две категории – детерминированные и недетерминированные. Детерминированное поведение функции зависит скорее не от того, каковы ее параметры, а от действий, выполняемых в самой функции. Если функция возвращает одно и то же значение после каждого вызова с одним и тем же набором допустимых параметров, эта функция называется *детерминированной*. Примером детерминированной встроенной функции может служить `SUM()`. Сумма чисел 3, 5 и 10, полученная с помощью этой функции, всегда равна 18. С другой стороны, функция `GETDATE()` является *недетерминированной*, поскольку возвращаемые ею результаты изменяются после каждого вызова.

Функция рассматривается как детерминированная, если она соответствует четырем описанным ниже критериям.

- Функция должна быть привязанной к схеме. Это означает, что все объекты, от которых зависит функция, должны иметь зарегистрированную зависимость и не допускается внесение изменений в определения этих объектов без предварительного уничтожения зависимой от них функции.
- Все другие функции, которые ссылаются на рассматриваемую функцию, также должны быть детерминированными, независимо от того, являются ли они определяемыми пользователем или определены в системе.
- В функции нельзя ссылаться на таблицы, которые определены вне самой функции (но использование переменных типа таблицы и временных таблиц допускается при условии, что эти переменные типа таблицы или временные таблицы объявлены в области определения функции).
- В функции нельзя применять расширенную хранимую процедуру.

В том, насколько важным является требование по обеспечению детерминированного выполнения, можно сразу же убедиться при попытке сформировать индекс на представлении или вычисленном столбце. Создание индексов на представлениях или вычисленных столбцах допускается, только если есть возможность надежно определить результат выборки данных из представления или вычисленного столбца. Это означает, что при наличии в представлении или вычисленном столбце ссылки на недетерминированную функцию не будет разрешено создание индекса на этом представлении или столбце. Безусловно, возникающая при этом ситуация не является безвыходной, но она вынуждает разработчика предпринимать дополнительные действия для определения того, является ли функция детерминированной или не детерминированной, прежде чем приступать к созданию индексов на представлениях или столбцах, в которых используется эта функция.

Это означает, что разработчику часто приходится решать важную задачу по определению того, является ли разрабатываемая им функция детерминированной или недетерминированной. При этом следует учитывать, что, кроме проверки соответствия создаваемой функции описанным выше критериям, можно также прибегнуть к помощи СУБД SQL Server, которая сообщает, является ли функция детерминированной или недетерминированной, поскольку информация об этом сохраняется в свойстве `IsDeterministic` интересующего вас объекта. Для проверки этой информации можно воспользоваться функцией `OBJECTPROPERTY`. Например, можно проверить детерминированность функции `DayOnly`, которая использовалась выше в данной главе, следующим образом:

```
USE Accounting
SELECT OBJECTPROPERTY(OBJECT_ID('DayOnly'), 'IsDeterministic')
```

Для многих программистов окажется неожиданным (а для многих, возможно, нет), что в ответе, полученном от СУБД, будет указано, что данная функция является недетерминированной:

```
-----------
0
(1 row(s) affected)
```

Проверьте указанный список критериев соответствия требованиям к детерминированной функции, чтобы узнать, сможете ли вы сами определить, почему данная функция не рассматривается как детерминированная.

Работая над этим примером, автор получил одно из тех не очень приятных напоминаний о том, насколько трудно обойтись без элементарных ошибок. Безусловно, я был уверен, что эта функция должна быть детерминированной, но она отнюдь не стала таковой по определению. Просидев за работой без сна слишком много ночей и уходя на отдых только в предрассветные часы, я полностью забыл выполнить очевидное требование – ввести опцию `WITH SCHEMABINDING`.

К счастью, существует возможность легко исправить тот единственный недостаток, который обнаруживается в определении функции `DayOnly`. Достаточно лишь ввести в определение функции опцию `WITH SCHEMABINDING`, и полученные результаты станут совсем другими:

```
ALTER FUNCTION DayOnly(@Date datetime)
RETURNS varchar(12)
WITH SCHEMABINDING
AS
BEGIN
  RETURN CONVERT(varchar(12), @Date, 101)
END
```

Теперь вызовем повторно на выполнение тот же запрос с функцией `OBJECTPROPERTY`, после чего будут получены данные, свидетельствующие о том, что теперь эта функция является детерминированной:

```
-----------
1
(1 row(s) affected)
```

Однако попытка применить такую же проверку к описанной выше функции `AveragePrice`, которая была создана в базе данных `pubs`, приводит к получению совсем других результатов. Эта функция выглядела примерно таким образом:

```
USE Pubs
GO
CREATE FUNCTION dbo.AveragePrice()
RETURNS money
WITH SCHEMABINDING
AS
BEGIN
  RETURN (SELECT AVG(Price) FROM dbo.Titles)
END
```

В функции `AveragePrice` привязка к схеме была осуществлена с самого начала. поэтому рассмотрим, какие результаты дает применение функции `OBJECTPROPERTY`:

```
SELECT OBJECTPROPERTY(OBJECT_ID('AveragePrice'), 'IsDeterministic')
```

Оказывается, что результаты выполнения функции OBJECTPROPERTY свидетельствуют о том, что функция AveragePrice является недетерминированной, даже несмотря на то, что она связана со схемой. Дело в том, что в данной функции применяется ссылка на таблицу, не являющуюся локальной по отношению к самой функции (на временную таблицу или на переменную типа таблицы, созданную вне функции).

Следует также отметить, что недетерминированной является и функция PriceDifference, описанная в том же разделе, что и функция AveragePrice. Безусловно, одна из причин этого заключается в том, что в определении функции PriceDifference не предусмотрено ее связывание со схемой, но еще важнее то, что в функции PriceDifference применяется ссылка на функцию AveragePrice. Как уже было сказано, если создаваемая функция ссылается на недетерминированную функцию, то по определению сама становится недетерминированной.

Отладка пользовательских функций

По существу, процесс отладки пользовательских функций весьма напоминает процесс отладки хранимых процедур, пример которого приведен в главе 12.

Откройте в программе Visual Studio окно Server Explorer, чтобы установить соединение и перейти к пользовательской функции. Щелкните правой кнопкой мыши на обозначении этой функции и выберите во всплывающем меню команду Step Into; выполняемые в ходе отладки действия являются полностью одинаковыми, за исключением того, что сам отлаживаемый программный объект берется из другого списка (из списка функций, а не хранимых процедур).

Применение инфраструктуры .NET для работы с базами данных

Как было описано в главе 12, в версии SQL Server 2005 предусмотрена возможность использовать сборки .NET в хранимых процедурах и функциях. Благодаря этому чрезвычайно расширились возможности не только хранимых процедур, но и функций.

Автор исходит из того, что большинство читателей настоящей книги относятся к категории начинающих разработчиков, поэтому полностью представляют себе, насколько трудно объяснить все последствия, связанные с применением возможностей инфраструктуры .NET для доступа к базам данных. В действительности прибегать к использованию всех открывающихся при этом возможностей приходится не слишком часто, но когда возникает такая необходимость, в большинстве случаев удается достичь весьма впечатляющих результатов. В частности, средства .NET позволяют реализовывать сложные формулы вычисления специальных алгоритмов, дают возможность обращаться к внешним источникам данных (например, к базам данных компаний, которые берут на себя обязанности по авторизации кредитных карточек и выполняют тому подобные функции), обеспечивают доступ к другим структурированным источникам данных, и т.д. Короче говоря, благодаря применению инфраструктуры .NET внезапно становятся относительно осуществимыми такие операции обработки данных, которые до сих пор было либо невозможно реализовать с помощью

программного обеспечения для баз данных, либо для этого приходилось выполнять чрезвычайно трудоемкую разработку.

Подтверждением сказанного может отчасти служить приведенный в данной главе пример реализации сложной формулы с помощью пользовательской функции. Еще одним направлением использования открывающихся возможностей может стать обработка табличных данных из внешних источников, скажем, представленных в формате CSV (Comma-Separated Value – значения, разделенные запятыми) или в каком-то подобном формате, с помощью сборки .NET, оформленной в виде функции СУБД SQL Server.

Однако вся тематика применения сборок .NET в СУБД SQL Server остается весьма сложной, поэтому дальнейшее ее описание будет продолжено в книге по SQL Server 2005 для профессионалов. Несмотря на сказанное, отметим, что в данной главе приведены важные и достаточно полные сведения, позволяющие успешно подготовиться к освоению этих новых и перспективных возможностей.

Резюме

Очевидно, что для описания пользовательских функций, приведенного в этой главе, не потребовалось вводить большой объем нового учебного материала. Фактически при создании пользовательских функций используются в основном такие же операторы и переменные, а также реализуются такие же процедуры кодирования, как и во время разработки сценариев и хранимых процедур. Тем не менее пользовательские функции открывают возможность реализации новых и очень привлекательных функциональных средств, которое не могли быть ранее осуществлены в СУБД SQL Server. Особенно важно то, что пользовательские функции позволяют намного расширить область применения программного обеспечения и даже допускают возможность их вложения непосредственно в запросы. К тому же с помощью пользовательских функций могут также формироваться параметризованные представления и динамически создаваемые таблицы.

В целом можно сделать вывод, что пользовательские функции относятся к числу наиболее перспективных программных средств, которые были введены в последних версиях SQL Server. Очевидно, что в одной короткой главе невозможно было раскрыть весь потенциал пользовательских функций, поэтому мне остается лишь выразить надежду на то, что читатель сумеет внести свой вклад в разработку новых способов их применения.

Упражнения

13.1. Повторно реализуйте сценарий `spTriangular`, рассматриваемый в главе 12, но на этот раз в виде функции, а не хранимой процедуры.

14

Транзакции и блокировки

Это – одна из тех глав, освоение материала которой становится ступенькой на пути к подлинному профессионализму. Безусловно, транзакции и блокировки – это те области функционирования баз данных, которые чаще всего трактуются разработчиками неправильно, несмотря на то, что сведения о них, изложенные в данной главе, нельзя назвать очень сложными. По крайней мере, даже после изучения представленной здесь вводной информации (однако, возможно, автор ошибается, считая ее таковой), читатель будет вполне подготовлен к профессиональной работе в области баз данных.

В настоящей главе рассматриваются перечисленные ниже темы.

- Общее определение понятия транзакции.
- Описание функционирования журналов и контрольных точек СУБД SQL Server.
- Основные сведения о блокировках.

В целом в данной главе показано, насколько тесно связаны друг с другом рассматриваемые в ней темы, а также дана информация о том, как свести к минимуму сложности, касающиеся применения транзакций и блокировок.

Основные сведения о транзакциях

Одним из основных свойств транзакции является **неразрывность**. Под этим подразумевается выполнение определенной последовательности действий как единого целого. Транзакции, выполняемые в базе данных, – это действия, осуществляемые с применением одного или нескольких операторов, которые реализуются по принципу "все или ничего".

В процессе обработки данных часто возникает необходимость обеспечить, чтобы после выполнения одной операции обязательно должна была быть выполнена дру-

гая операция и т.д. либо чтобы все эти операции должны были окончиться неудачей. Фактически подобная цепь взаимосвязанных операций может возрастать до такой степени, что до двадцати (или больше) действий должны происходить в связи друг с другом или не должно быть выполнено ни одно из этих действий. Рассмотрим классический пример.

Клиент с идентификатором `Sally` приходит в банк и просит перечислить одну тысячу долларов со своего расчетного счета на свой сберегательный счет.

Для осуществления этой операции в базе данных достаточно применить следующие два оператора:

```
UPDATE checking
   SET Balance = Balance - 1000
   WHERE Account = 'Sally'
UPDATE savings
   SET Balance = Balance + 1000
   WHERE Account = 'Sally'
```

Это — предельно упрощенный пример, показывающий, какие операции выполняются при перечислении денег с расчетного счета (таблица `checking`) на сберегательный счет (таблица `savings`), но он подчеркивает основной смысл этих действий: должны быть выполнены две разные операции, по одной для каждого счета.

Теперь предположим, что первая из этих операций будет выполнена, а вторая — нет. В таком случае клиент `Sally` лишится тысячи долларов, а банк — своей репутации.

Поэтому при выполнении подобных взаимосвязанных операций необходимо обеспечить вслед за первой операцией осуществление второй операции и т.д. Сами операции базы данных фактически не позволяют достичь указанной цели, поскольку происходят одна за другой, и на каждом из этих этапов работы могут возникать всевозможные сбои, начиная с отказа аппаратных средств и заканчивая такими весьма вероятными ситуациями, как нарушение правил целостности данных. Но, к счастью, предусмотрена возможность применения некоторых программных конструкций, которые решают общую задачу по обеспечению выполнения всех взаимосвязанных действий, поэтому фактически не требуется контролировать вручную выполнение каждого отдельного оператора. Включив все операции в одну **транзакцию**, можно обеспечить, чтобы после неудачного завершения любой из операций были отменены результаты всех операций, выполненных до сих пор в составе этой транзакции.

Прежде всего, транзакция характеризуется явно определенными границами. Каждая транзакция должна иметь четко обозначенные точки начала и конца. По существу, как неявно заданная транзакция рассматривается и каждый отдельно взятый оператор `SELECT`, `INSERT`, `UPDATE` и `DELETE`, выполняемый в СУБД SQL Server. Причина, по которой отдельно взятый оператор рассматривается как транзакция, состоит в том, что любая операция должна осуществляться в базе данных по принципу "все или ничего", ведь нельзя, например, допустить, чтобы была удалена только часть строк, указанных в операторе удаления. Поэтому по умолчанию минимальная длина транзакции равна одному оператору.

Но иногда возникает необходимость выполнить по принципу "все или ничего" несколько операторов, например, как в описанной выше банковской операции с переводом денег со счета на счет. Это означает, что должен быть предусмотрен способ обозначения начала и конца транзакции, а также способ регламентации действий, которые должны производиться в случае успешного или неудачного завершения

транзакции. Для этого предусмотрено несколько описанных ниже операторов T-SQL, которые позволяют обозначить в транзакции указанные точки.

- Оператор BEGIN TRAN. Задает начальную точку транзакции.
- Оператор COMMIT TRAN. После выполнения этого оператора результаты осуществления транзакции становятся постоянной, неотъемлемой частью базы данных (это называется *фиксацией*).
- Оператор ROLLBACK TRAN. Выполнение этого оператора приводит к полной отмене (откату) изменений, внесенных в базу данных предыдущими операторами транзакции.
- Оператор SAVE TRAN. Устанавливает определенную отметку, которая позволяет произвести только частичный откат (такая отметка называется *контрольной точкой*).

Рассмотрим вначале отдельные операторы управления транзакциями, а после этого перейдем к описанию примеров транзакций.

Оператор BEGIN TRAN

По-видимому, действие оператора BEGIN TRAN, с которого начинается выполнение транзакции, можно объяснить проще всего. Единственное назначение этого оператора состоит в том, что он указывает, где начинается неразрывная единица работы, рассматриваемая как одна транзакция. Если по какой-то причине не удастся зафиксировать транзакцию или сложатся такие условия, при которых фиксация транзакции потеряет смысл, то отменяются все действия, выполненные в базе данных в составе транзакции, и база данных возвращается в то состояние, которое она имела в точке вызова этого оператора (происходит откат). Иными словами, все результаты действий, выполненных в составе транзакции, начиная с этой точки, которые не были в конечном итоге зафиксированы, фактически отменяются (безусловно, только в той части. которая касается самой базы данных).

Оператор BEGIN TRAN имеет следующий синтаксис:

```
BEGIN TRAN[SACTION] [<transaction name>|<@transaction variable>]
```

Оператор COMMIT TRAN

Оператор COMMIT TRAN обеспечивает фиксацию транзакции и тем самым обозначает конец выполнения транзакции. С момента выполнения оператора COMMIT TRAN транзакция, к которой относится этот оператор, переходит в категорию **устойчивых**. Иными словами, с этого момента результаты выполнения транзакции становятся постоянными и сохраняются даже после аварийного завершения работы системы (но разумеется, это не гарантирует сохранения данных в случае физического уничтожения файлов базы данных, если отсутствует их резервная копия). Единственным способом отмены изменений, внесенных в базу данных в результате выполнения зафиксированной транзакции, является вызов новой транзакции, которая осуществила бы действия, обратные по отношению к предыдущей транзакции.

Синтаксис оператора COMMIT TRAN во многом напоминает синтаксис оператора BEGIN TRAN:

```
COMMIT TRAN[SACTION] [<transaction name>|<@transaction variable>]
```

Оператор ROLLBACK TRAN

Наглядным описанием операции отката, выполняемой с помощью оператора ROLLBACK TRAN, может стать совет, содержащийся в словах песни "Начни сначала". Иначе говоря, если что-то не удается успешно довести до конца, то лучше всего отменить сделанное и вернуться к исходной точке.

Обычно на практике такая рекомендация вполне оправдывается. А достичь именно этой цели — вернуться в состояние, которое имело место до начала транзакции, — позволяет оператор ROLLBACK TRAN. Тем не менее происходит отмена только действий, выполненных в транзакции, к которой относится оператор ROLLBACK TRAN. При этом исключаются все изменения, которые произошли в базе данных в результате действий, выполненных в составе транзакции со времени вызова оператора BEGIN TRAN этой транзакции. Но если в ходе осуществления транзакции были установлены так называемые **точки сохранения**, которые вскоре будут описаны, то отмена выполнения происходит до одной из предыдущих точек сохранения.

Оператор ROLLBACK TRAN имеет в основном такой же синтаксис, как и другие описанные выше операторы управления транзакциями, не считая того, что в нем предусмотрена также возможность указать точку сохранения:

```
ROLLBACK TRAN[SACTION] [<transaction name>|<save point name>|
   <@transaction variable>|<@savepoint variable>]
```

Оператор SAVE TRAN

Промежуточное сохранение результатов транзакции с помощью оператора SAVE TRAN представляет собой создание своего рода вспомогательного объекта, называемого *точкой сохранения*, который напоминает закладку в недочитанной книге. Точке сохранения присваивается имя (что позволяет отличить ее от других точек сохранения). После создания точка сохранения может быть указана при выполнении отката. Такая возможность является очень удобной, поскольку позволяет точно определить в своем коде, к какому состоянию базы данных необходимо вернуться в том случае, если будет обнаружено нарушение в работе. Для этого достаточно лишь указать точку сохранения, к которой должен быть выполнен откат.

Оператор SAVE TRAN имеет следующий довольно простой синтаксис:

```
SAVE TRAN[SACTION] [<save point name>| <@savepoint variable>]
```

При использовании точек сохранения необходимо учитывать то, что после выполнения оператора ROLLBACK TRAN все они уничтожаются. Это означает, что даже если будет создано пять точек сохранения, все они исчезают после выполнения единственного оператора ROLLBACK TRAN. Безусловно, возможность снова приступить к созданию новых точек сохранения не исключается, так же как остается возможность выполнить откат к новым точкам сохранения, но все точки сохранения, созданные ко времени выполнения последнего оператора ROLLBACK TRAN, уничтожаются.

Очевидно, что при использовании оператора SAVE TRAN могут возникать чрезвычайно запутанные ситуации, поэтому автор не может порекомендовать его применение на первых порах освоения работы с базой данных, но следует помнить о том, что он существует.

Принципы функционирования журналов СУБД SQL Server

Автор, безусловно, был обязан привести основные сведения о транзакциях, прежде чем приступить к описанию способов, применяемых в СУБД SQL Server для отслеживания содержимого базы данных. Дело в том, что в процессе работы объект, называемый базой данных, почти никогда не содержит в своих файлах последнюю и окончательную версию всех данных. Необходимо учитывать, что в процессе эксплуатации информация базы данных состоит не только из той информации, которая хранится в физическом файле (файлах) базы данных, но и включает в себя промежуточную информацию, накапливаемую в ходе осуществления всех транзакций, которая записывается в журнал базы данных, начиная с последней контрольной точки. После фиксации транзакций информация из журнала переносится в базу данных, но в процессе работы такая ситуация, когда в журнале отсутствует информация, подлежащая перезаписи в базу данных, встречается редко.

Обычно в ходе эксплуатации базы данных сведения о выполнении большинства действий с данными вначале регистрируются в **журнале транзакций**, а не записываются непосредственно в базу данных. Журнал ведется в оперативной памяти. Он представляет собой кэш, состоящий из отдельных страниц, в которые из базы данных считываются данные, подлежащие модификации. Создание **контрольной точки** — это промежуточная операция, выполнение которой приводит к перезаписи на диск из оперативной памяти всех модифицированных, но незафиксированных страниц, используемых в настоящее время для работы с базой данных. *Незафиксированными страницами* называются страницы журнала или страницы данных, в которые были внесены изменения после того, как они были прочитаны в кэш, но соответствующие модифицированные данные еще не были записаны на диск. Если не используются контрольные точки, то возникает опасность, что журнал выйдет за пределы допустимых размеров и (или) заполнит все имеющееся дисковое пространство. Блок-схема процесса ведения журнала показана на рис. 14.1.

Не следует рассматривать приведенные выше сведения об использовании кэша как указание на то, что программист обязан предусматривать в коде какие-то специальные действия для вывода данных из кэша. Все операции, связанные с использованием кэша, осуществляются СУБД SQL Server автоматически. А приведенное здесь описание предназначено для использования только в качестве иллюстрации, позволяющей понять, как применяются журналы и какие действия требуются для осуществления транзакции. Кроме того, если требуется достичь максимальной производительности, то необходимо руководствоваться сведениями о том, как данные передаются в кэш и как долго они хранятся на страницах кэша в оперативной памяти, поэтому важно знать, как происходит запись в журнал, как данные попадают в кэш и выводятся из него.

Необходимо отметить, что контрольные точки применяются не только в самой базе данных для освобождения кэша, в которой должны быть считаны новые данные, но и в связи с выполнением операций, описанных ниже.

- Очистка кэша вручную с помощью команды `CHECKPOINT`.
- Нормальный останов сервера (останов, в ходе которого не используется опция `WITH NOWAIT`, отменяющая выгрузку кэша на диск).

- Изменение любого аспекта режима эксплуатации базы данных (например, переход к монопольному применению базы данных, к эксплуатации только под управлением владельца базы данных, `dbo`, и т.д.).
- Применение опции `Simple Recovery` после заполнения журнала на 70%.
- Выгрузка данных после увеличения объема данных в журнале (со времени применения последней контрольной точки) сверх той величины, которая может быть восстановлена сервером за время, указанное в опции, называемой **интервалом восстановления** (эта часть журнала называется **активной**).

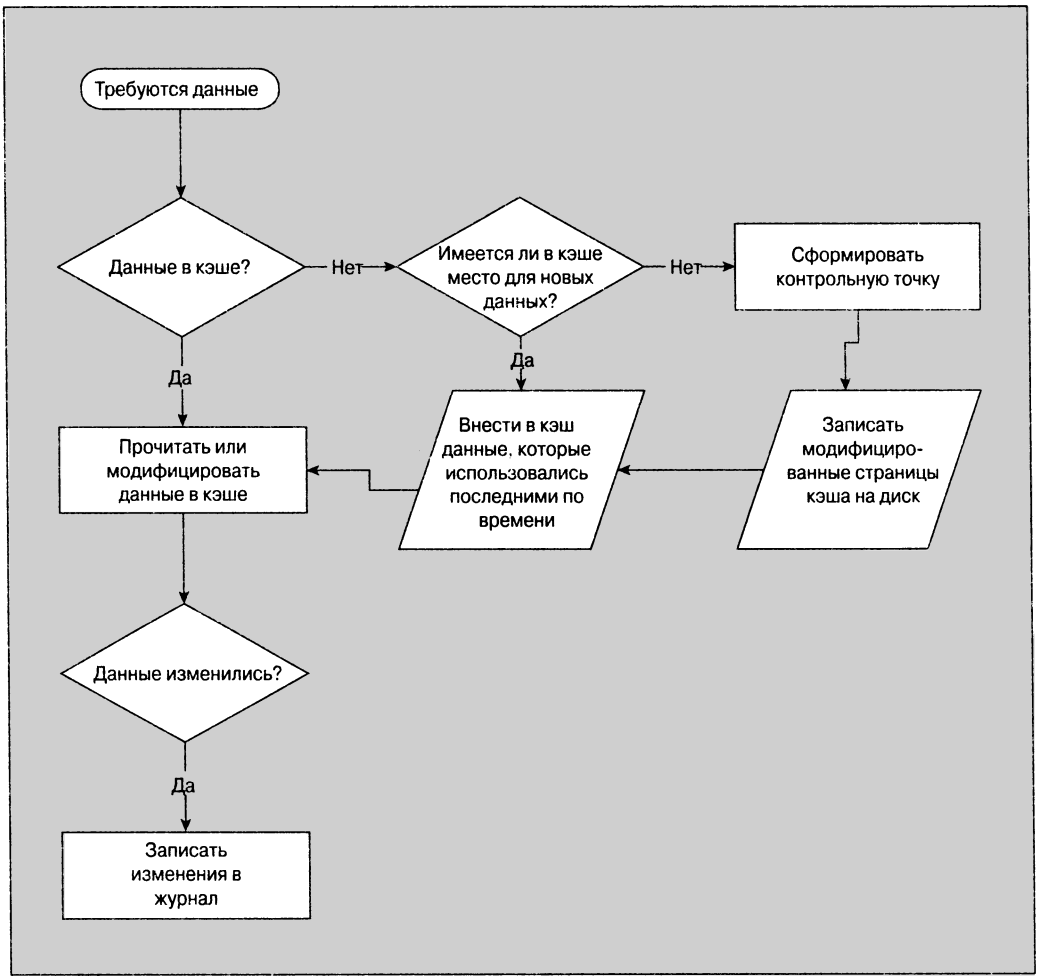


Рис. 14.1. Блок-схема процесса ведения журнала

Аварийный отказ и восстановление

Фактически восстановление баз данных происходит при каждом запуске СУБД SQL Server. Восстановление сводится к тому, что открывается файл базы данных, а затем в базу данных вносятся все зафиксированные изменения, которые были отражены в журнале со времени применения последней контрольной точки (изменения вносятся путем их записи в физические файлы базы данных). Применительно ко всем зарегистрированным в журнале изменениям, за которыми не следует соответствующая операция фиксации, выполняется откат. Иными словами, при этом по сути исключается вся информация о том, что когда-либо происходили такие изменения.

Рассмотрим осуществление указанных действий на примере выполнения нескольких транзакций в базе данных. Допустим, что в журнале содержится информация о пяти транзакциях (рис. 14.2).

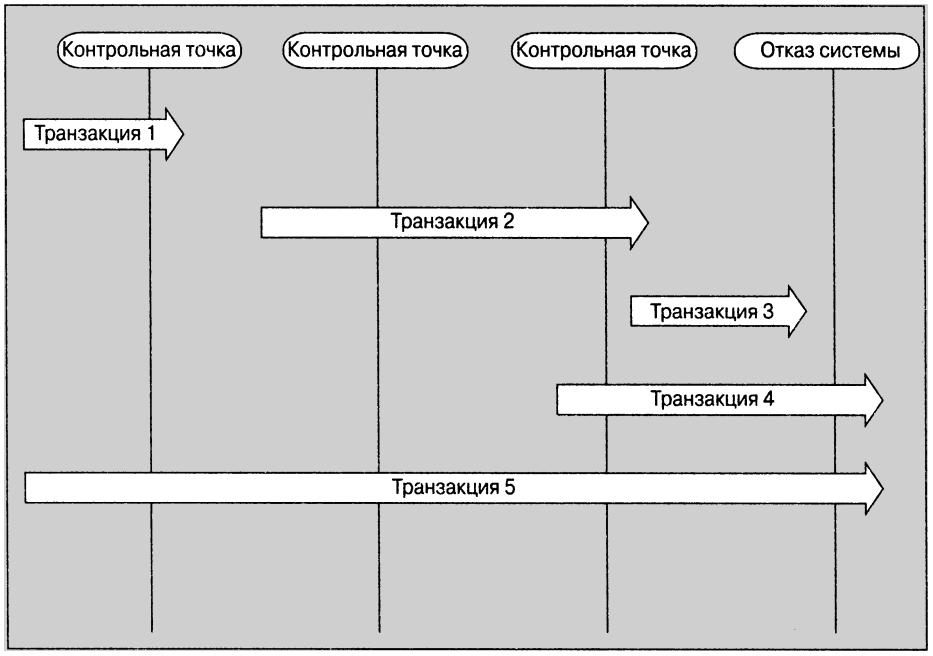


Рис. 14.2. Схематическое представление результатов регистрации пяти транзакций в журнале

Ниже приведено последовательное описание того, что происходит в каждой из этих транзакций.

Транзакция 1

На этапе восстановления применительно к транзакции 1 не будет выполнено никаких действий. После фиксации этой транзакции уже была применена одна контрольная точка, поэтому все изменения, осуществляемые в этой транзакции, уже были полностью зафиксированы в базе данных. В ходе восстановления применительно к транзакции 1 не требуется выполнения каких-либо действий, а данные, которые

будут считываться в кэш в ходе дальнейшего выполнения операций обработки данных, должны отражать результаты всех зафиксированных транзакций.

Транзакция 2

Ко времени применения последней контрольной точки транзакция 2 не была завершена, поэтому ее фиксация еще не произошла (иными словами, выполнение транзакции 2 продолжалось). Но транзакция, для которой не выполнена фиксация, фактически не рассматривается в процессе применения контрольной точки. Поэтому должен быть выполнен так называемый *накат* этой транзакции. Выполнение операции наката сводится к тому, что все страницы, связанные с выполнением транзакции, снова считываются в кэш, а информация из журналов применяется для повторного выполнения всех операторов, применяемых в составе данной транзакции. После завершения этих действий состояние транзакции 2 должно полностью соответствовать тому, в котором она находилась до аварийного завершения работы системы.

Транзакция 3

На первый взгляд может создаться другое впечатление, но действия, которые должны быть осуществлены при восстановлении базы данных, полностью соответствуют тем, которые связаны с транзакцией 2. И в данном случае транзакция 3 не была завершена во время применения последней контрольной точки, т.е. эта транзакция не принимала участие в осуществлении контрольной точки, как и транзакция 2. Единственное отличие состоит в том, что к этому времени транзакция 3 даже не существовала, но с точки зрения процесса восстановления это не имеет большого значения, поскольку подход к восстановлению данных, модифицированных в результате выполнения транзакции, зависит от того, была ли выполнена фиксация этой транзакции.

Транзакция 4

Транзакция 4 не была завершена ко времени возникновения аварийного отказа системы, поэтому должен быть выполнен ее откат. По существу, все изменения, внесенные в ходе этой транзакции, должны быть полностью отменены. Это означает, что пользователь должен повторно ввести все данные, относящиеся к этой транзакции, и все связанные с ней процессы обработки данных необходимо начать сначала.

Транзакция 5

Транзакция 5 в основном не отличается от транзакции 4. Различия между транзакциями этих двух типов являются чисто внешними и обусловлены тем, что транзакция 5 осуществлялась в течение более продолжительного времени, но суть действий по восстановлению от этого не меняется. Транзакция не была зафиксирована ко времени аварийного останова системы, поэтому для нее должен быть выполнен откат.

Неявные транзакции

В СУБД SQL Server в основном для обеспечения совместимости с другими важными системами реляционных СУБД, такими как Oracle или DB2, предусмотрена возможность организации работы с помощью так называемых **неявных транзакций**. По умолчанию поддержка неявных транзакций в СУБД SQL Server не применяется, но

может быть введена в действие. В режиме работы с использованием неявных транзакций не требуются операторы BEGIN TRAN, поскольку в качестве оператора начала транзакции рассматривается первый оператор, вызываемый на выполнение в базе данных, и транзакция запускается автоматически. Неявная транзакция осуществляется до тех пор, пока на выполнение не будет вызван оператор COMMIT TRAN или ROLLBACK TRAN. Затем со следующего оператора начинается очередная транзакция.

Организация работы с использованием неявных транзакций может быть связана с возникновением всевозможных осложнений, поэтому ее описание выходит за рамки рассмотрения настоящей книги. Достаточно сказать, что опцию поддержки неявных транзакций рекомендуется использовать лишь в тех случаях, когда на это есть очень важные причины (например, требуется обеспечить совместимость с программным обеспечением, написанным для другой системы).

Блокировки и параллельная организация работы

В любой системе баз данных очень важно обеспечить **параллельную организацию работы**. Для этого прежде всего необходимо найти способ обеспечения доступа для двух или большего числа пользователей одновременно к одному и тому же объекту. Безусловно, при этом разным пользователям может потребоваться осуществлять разные действия (обновление, удаление, чтение и вставку данных), поэтому, чтобы добиться идеального функционирования базы данных, необходимо найти способ устранения конфликтов между операциями, выполняемыми всеми пользователями, с учетом того, какие действия они осуществляют и насколько важны эти действия. Если СУБД обеспечивает одновременное выполнение многочисленных операций, то производительность базы данных по мере увеличения количества пользователей (точнее, количества одновременно выполняемых транзакций) не уменьшается, а увеличивается.

Как правило, в среде OLTP (OnLine Transaction Processing – оперативная обработка транзакций) наиболее важным требованием к организации работы является обеспечение параллельного выполнения операций. Именно на этом основана структура большинства объектов баз данных, которые рассматриваются в настоящей книге. С другой стороны, в среде OLAP (Online Analytical Processing – оперативная аналитическая обработка данных) проблема обеспечения максимальной производительности за счет параллельной организации работы в основном теряет свою остроту, хотя во многих системах ее нельзя полностью переводить на второй план. В целом следует отметить, что без параллельной организации работы невозможно добиться приемлемой производительности системы. Основой параллельной организации работы в базах данных являются операции, называемые **блокировками**.

Блокировки позволяют предотвратить применение к объекту базы данных таких операций, которые конфликтуют с операциями, уже выполняемыми над этим объектом. При этом учитывается то, какая из операций, связанных с доступом к объекту, началась раньше. Действия, которые могут и не могут быть выполнены в операциях, начавшихся позже, зависят от того, какие действия осуществляются в операции, начавшейся раньше. Это означает, что информация о производимых действиях должна быть доступна системе, чтобы можно было определить, совместимы ли действия, предусмотренные в начавшемся позже процессе, с действиями ранее начатого про-

цесса. Например, обычно разрешается доступ к одному и тому же фрагменту данных для одного, двух, десяти, ста, тысячи или любого количества пользователей, подключившихся к системе, при условии, что все эти пользователи обращаются к соответствующей строке таблицы в режиме только чтения. В данном случае напрашивается сравнение с выставкой-продажей картин — многочисленные посетители могут рассматривать экспонаты, часто одни и те же, и такая возможность остается до тех пор, пока выставка не переезжает, на ней остаются непроданные картины или не происходят какие-то другие изменения. При этом передача в руки одной и той же картины больше чем одному лицу не допускается. Именно поэтому организаторы выставки внимательно отслеживают пожелания посетителей и часто вынуждены решать, чья заявка на покупку должна быть выполнена в первую очередь.

Роль такого организатора выставки выполняет **диспетчер блокировок** СУБД SQL Server. После того как пользователь обращается с запросом к СУБД SQL Server, диспетчер блокировок распознает назначение этого запроса, чтобы определить, какие действия должны быть связаны с его выполнением. Если в запросе речь идет только о просмотре данных и все прочие запросы пользователей предназначены исключительно для просмотра, то диспетчер блокировок позволяет выполнить операцию выборки данных еще одному пользователю. Если же в намерения пользователя входит внесение каких-либо изменений (обновление или удаление), то диспетчер блокировок проверяет, обращается ли еще кто-либо к требуемому фрагменту данных. В случае положительного ответа текущий пользователь должен ожидать завершения предыдущей операции, и это же относится ко всем, кто попытался обратиться к тем же данным после него. Дело в том, что операции модификации данных может проводить одновременно только один пользователь, поэтому все прочие пользователи должны ожидать завершения этих операций.

Благодаря тому что доступ к данным в СУБД SQL Server организован таким образом, появляется возможность избежать возникновения многочисленных и разнообразных проблем, которые могли быть обусловлены неправильной организацией параллельного доступа. В настоящей главе рассматриваются возможные проблемы, связанные с обеспечением параллельной работы, и показано, как выбрать **уровень изоляции транзакции**, позволяющий предотвратить каждую из этих проблем, но на данном этапе рассмотрим, для чего могут и не могут применяться блокировки и к каким типам относятся доступные блокировки.

Возможные нарушения в работе, предотвращаемые с помощью блокировок

Блокировки позволяют устранить четыре описанных ниже нарушения в работе.

- Чтение незафиксированных данных.
- Неповторяемое чтение.
- Фантомы.
- Потерянные обновления.

С каждым из этих нарушений в работе связано отдельное множество проблем, но со всеми потенциальными ошибками позволяет справиться ряд решений, которые обычно предусматривают правильный выбор уровня изоляции транзакции. Информация, приведенная в этой главе, потребуется при изучении следующей главы. В состав этой информации входят сведения о том, какой уровень изоляции транзак-

ции является наиболее приемлемым для решения каждой из рассматриваемых проблем. Более полное описание уровней изоляции транзакции будет вскоре приведено, а пока кратко рассмотрим суть каждого из указанных нарушений в работе.

Чтение незафиксированных данных

Чтение незафиксированных данных происходит, если в транзакции считывается строка, модифицируемая в другой транзакции, которая еще не завершена. Если транзакция, начатая первой, завершится успешно, то, по-видимому, внесенные в ней изменения останутся неизменными и проблема не возникнет. А если произойдет откат транзакции, в которой были модифицированы данные, считанные в другой транзакции, то окажется, что считаны данные, которые фактически отсутствуют в базе данных!

Рассмотрим на примере, показанном в табл. 14.1, как возникает ошибка, связанная с чтением незафиксированных данных.

Таблица 14.1. Пример того, как происходит чтение незафиксированных данных

Оператор транзакции 1	Оператор транзакции 2	Исходное значение данных	Незафиксированное значение данных	Значение, обнаруживаемое в транзакции 2
`BEGIN TRAN`		3		
`UPDATE col = 5`	`BEGIN TRAN`	3	5	
`SELECT anything`	`SELECT @var = col`	3	5	5
`ROLLBACK`	`UPDATE anything SET whatever = @var`		3	5

В табл. 14.1 наглядно показано, как возникает проблема.

Но с этого времени в транзакции 2 используется недействительное значение. А при попытке повторить эту ситуацию и проследить выполняемые в базе данных действия, чтобы понять, как получено неправильное значение, обнаруживается, что какая-либо информация трассировки отсутствует, поэтому отладка приложения становится чрезвычайно затруднительной.

К счастью, такое развитие событий невозможно, если используется заданный в СУБД SQL Server по умолчанию уровень изоляции транзакции (этот уровень изоляции, называемый `READ COMMITTED`, будет описан более подробно в разделе "Определение уровня изоляции транзакции").

Неповторяемое чтение

Нарушение в работе, связанное с неповторяемым чтением, фактически можно легко спутать с чтением незафиксированных данных. Но после ознакомления с определениями и конкретными примерами суть рассматриваемых понятий становится ясной, и путаница в терминологии исключается.

Неповторяемое чтение происходит, если строка считывается в транзакции дважды, а между тем происходит модификация считываемых данных в отдельной транзакции. В качестве иллюстрации данного нарушения в работе снова рассмотрим пример с банком. В табл. 14.2 показано возможное развитие ситуации при возникновении неповторяемого чтения. Отметим, что остаток на банковском счете не должен быть отрицательным.

Таблица 14.2. Возможное развитие ситуации при возникновении неповторяемого чтения

Транзакция 1	Транзакция 2	Значение переменной @Var	Данные таблицы, полученные в транзакции 1	Значение, фактически находящееся в таблице
BEGIN TRAN		NULL		125
SELECT @Var = value FROM table	BEGIN TRAN	125	125	125
	UPDATE value, SET value = value - 50			75
IF @Var >=100	END TRAN	125	125	75
UPDATE value, SET value = value - 100		125	125 (ожидание освобождения блокировки)	75
(Завершение операции, ожидание освобождения блокировки, а затем продолжение работы.)		125	75	Вариант 1: -25 (если не задано ограничение CHECK, не допускающее появления отрицательных значений). Вариант 2: ошибка с кодом 547 (если ограничение CHECK задано)

Итак, снова возникает проблема. В транзакции 1 было выполнено предварительное считывание данных (а в некоторых случаях рекомендуется именно такая организация работы приложения, как указано в разделе "Обработка ошибок еще до того, как они происходят" главы 12) для проверки того, что обрабатываемое значение является действительным, после чего выполнение транзакции продолжилось (поскольку количество денег на счете является достаточным). Но проблема заключается в том, что еще до выполнения оператора UPDATE в транзакции 1 удается выполнить в СУБД операцию модификации данных, относящуюся к транзакции 2. Если на таблице не задано ограничение CHECK, позволяющее предотвратить появление в столбце отрицательного значения, то остатку на счете действительно будет присвоено значение -25, даже несмотря на то, что это противоречит здравому смыслу, поскольку такая возможность должна была быть предотвращена благодаря использованию оператора IF в транзакции 1.

Возникновение указанной проблемы может быть исключено с помощью только двух описанных ниже способов.

- Создание ограничения CHECK и контроль за возникновением ошибки с кодом 547.
- Применение параметра ISOLATION LEVEL, имеющего значение REPEATABLE READ или SERIALIZABLE.

На первый взгляд кажется, что наиболее приемлемое решение состоит в использовании ограничения CHECK. Однако следует подчеркнуть, что такой метод устранения указанного нарушения в работе основан, скорее, на осуществлении действий, представляющих собой просто реакцию на сложившуюся ситуацию, а не на использовании превентивного подхода. Но в большинстве ситуаций вероятность неповто-

ряемого чтения достаточно велика, поэтому автор рекомендует использовать именно этот метод.

Более полное описание уровней изоляции транзакции будет приведено ниже, а пока достаточно отметить, что при использовании значений уровней изоляции REPEATABLE READ и SERIALIZABLE велика вероятность возникновения большего количества проблем, чем может быть решено с их помощью. Тем не менее возможность применения этих значений исключать нельзя.

Фантомы

Разумеется, в этом контексте речь идет не о тех фантомах, которым посвящены фильмы ужасов. В терминологии СУБД *фантомами* называются строки, которые появляются и исчезают загадочным образом, как будто они не подчиняются применяемым в базе данных операторам UPDATE и DELETE. Но иногда появление фантомов вполне можно предвидеть в процессе обычной эксплуатации системы, и подобную ситуацию отнюдь не сложно проиллюстрировать. Ниже описан классический пример появления фантома.

Предположим, что в базе данных представлена информация о служащих ресторана быстрого питания. На подобных предприятиях обычно имеется довольно значительное количество служащих, труд которых оплачивается по минимальной заработной плате, определенной правительством. Правительство только что выпустило постановление о повышении минимальной заработной платы с 6,25 до 6,75 доллара в час, поэтому необходимо обновить таблицу Employees, чтобы перевести всех тех, кто получает меньше 6,75 доллара в час, на новую минимальную заработную плату. Безусловно, программист не предвидит возникновения каких-либо сложностей и передает на выполнение следующий довольно простой оператор:

```
UPDATE Employees
SET HourlyRate = 6.75
WHERE HourlyRate < 6.75
ALTER TABLE Employees
   ADD ckWage CHECK (HourlyRate >= 6.75)
GO
```

Сумеет ли читатель обнаружить, в чем состоит ошибка, ведь необходимо отметить, что при попытке выполнить этот оператор появляется следующее сообщение об ошибке:

```
Msg 547, Level 16, State 1, Line 1
ALTER TABLE statement conflicted with COLUMN CHECK constraint 'ckWage'.
The conflict occurred in database 'FastFood', table 'Employees', column
'HourlyRate'.
```

После этого достаточно применить простой оператор SELECT для проверки наличия значений меньше 6.75 и убедиться в том, что такое значение действительно имеется. Но все же остается непонятным, как такое может случиться, ведь был предусмотрен оператор UPDATE, который должен был исключить подобную возможность. Действительно, оператор был выполнен и его выполнение прошло вполне успешно, а дело лишь в том, что появился **фантом**.

Примеры фантомного чтения встречаются редко, и для возникновения подобных ситуаций требуется определенное стечение обстоятельств. Кратко можно отметить, что именно в тот период, когда выполнялся оператор UPDATE, какой-то другой поль-

зователь провел операцию вставки данных с помощью оператора INSERT. А поскольку эта строка оказалась полностью новой, она не была охвачена блокировкой, поэтому ее вставка осуществлена вполне успешно.

Единственный способ предотвращения возможности возникновения подобных обстоятельств состоит в использовании значения уровня изоляции транзакции, равного SERIALIZABLE. При использовании этого значения любые другие операторы обновления таблицы должны выполняться в соответствии с условиями, которые не противоречат условиям, заданным в конструкции WHERE текущего оператора, так как в противном случае они будут заблокированы.

Потерянное обновление

Ситуация **потерянного обновления** возникает, если результаты одного обновления успешно записываются в базу данных, но по стечению обстоятельств эти результаты перекрываются другой транзакцией. Но на первый взгляд трудно представить себе, как такое может произойти.

Потерянные обновления могут возникать, если в двух транзакциях считывается вся строка, затем в одной из них в строку записывается обновленная информация, а вслед за этим запись обновленной информации в ту же строку происходит в другой транзакции. Рассмотрим пример такой ситуации.

Предположим, что аналитик отдела кредитования компании получил заявку с информацией о том, что заказчик X получил кредит на сумму, приближающуюся к лимиту кредитования, но хотел бы взять дополнительный кредит. Аналитик извлекает из базы данных информацию об этом заказчике, чтобы ознакомиться с ней. Обнаруживается, что лимит кредита составляет 5 тысяч долларов, но этот заказчик всегда погашает кредит вовремя.

Между тем другой сотрудник отдела кредитования извлекает строку с данными заказчика X, чтобы внести изменения в поле с его адресом. В полученной строке также содержится значение лимита кредитования, равное 5 тысячам долларов

К этому времени первый сотрудник принимает решение удовлетворить запрос и повысить лимит кредитования заказчика X до 7500 долларов. Он вводит это значение и нажимает клавишу <Enter>. С данного момента в базе данных находится значение лимита кредитования для заказчика X, равное 7500 долларам.

После этого второй сотрудник заканчивает корректировку адреса, но поскольку он использует такую же форму редактирования, что и первый сотрудник, после сохранения в базе данных откорректированной информации происходит обновление всей строки. Напомним, что в форме редактирования второго сотрудника находилось значение лимита кредитования, равное 5 тысячам долларов. Таким образом, в базе данных снова находится значение лимита кредитования заказчика X, равное 5 тысячам долларов, а обновление, внесенное первым сотрудником, потеряно.

Решение указанной проблемы должно быть основано на том. чтобы каким-то образом обеспечить возможность определения в коде, что происходит корректировка строки в промежутке между чтением данных и их обновлением. Выбор способа распознавания такой ситуации зависит от используемого метода доступа.

Блокируемые ресурсы

В СУБД SQL Server предусмотрено шесть различных типов **блокируемых ресурсов**, которые образуют иерархию. Чем выше уровень блокировки, тем крупнее ее

степень детализации (иными словами, при повышении уровня блокировки происходит выбор все большего и большего количества блокируемых объектов, на которые может распространяться любое каскадно выполняемое действие, даже лишь потому, что заблокирован сам объект, к которому относится это действие). Блокируемые ресурсы, расположенные в порядке возрастания степени детализации, описаны ниже.

- База данных. Если в качестве блокируемого ресурса рассматривается база данных, то происходит блокировка всей базы данных. Обычно такая степень детализации блокировки применяется при внесении изменений в схему базы данных.
- Таблица. Если выбрана степень детализации блокировки на уровне таблицы, то блокируется вся таблица. К этому относятся все объекты данных, связанные с самой таблицей, в том числе действительные строки с данными (все строки без исключения) и все ключи, определенные на всех индексах, связанных с рассматриваемой таблицей.
- Экстент. Если выбрана степень детализации блокировки на уровне экстента, то блокируется весь экстент. Напомним, что экстент состоит из восьми страниц, поэтому блокировка экстента предусматривает получение в результате блокировки под свой контроль всего экстента, восьми страниц данных или индекса в этом экстенте, а также всех строк с данными на этих восьми страницах.
- Страница. Применение степени детализации блокировки на уровне страницы приводит к тому, что блокируются все данные или ключи индекса на этой странице.
- Ключ. Блокировка со степенью детализации на уровне ключа представляет собой блокировку конкретного ключа или ряда ключей в индексе. Другие ключи, заданные на той же странице индекса, могут оказаться незатронутыми.
- Строка или идентификатор строки (Row Identifier – RID). Формально при использовании блокировки со степенью детализации на уровне строки считается, что блокировка распространяется на идентификатор строки (это – внутренняя конструкция СУБД SQL Server), но по существу блокируется вся строка.

Процесс эскалации блокировок и влияние блокировок на производительность

Стремление использовать более высокую степень детализации блокировки (допустим, поддерживать блокировку на уровне строки, а не страницы) оправдано, если количество блокируемых элементов невелико, но по мере увеличения количества блокируемых элементов издержки на поддержание всех этих блокировок возрастают настолько значительно, что производительность начинает заметно снижаться. Это означает, что блокировки в связи с уменьшением скорости обработки приходится оставлять на более продолжительное время, а это приводит к увеличению конкуренции за ресурсы, ведь чем дольше строка остается заблокированной, тем выше вероятность того, что эта строка потребуется кому-то другому. Чтобы выйти из этой ситуации, можно заменить несколько блокировок с мелкой степенью детализации одной блокировкой с более крупной степенью детализации; такой процесс объединения блокировок называется **эскалацией блокировок**. При выборе оптимальной степени детализации блокировок необходимо найти компромисс, не допуская чрезмерного

увеличения количества блокировок и, с другой стороны, не применяя блокировки, которые охватывают слишком крупные объекты. Если же в ходе работы количество блокировок существенно увеличивается, то в диспетчере блокировок применяется эскалация для перехода на следующий уровень.

После того как количество поддерживаемых блокировок достигает определенного порогового значения, происходит эскалация блокировок и переход на следующий, более высокий уровень детализации, что позволяет уменьшить объем работы, связанный с обслуживанием блокировок более низкого уровня (в результате освобождаются ресурсы, скорость обработки данных повышается и конкуренция за ресурсы становится менее острой).

Следует отметить, что решение об эскалации уровня блокировок принимается с учетом количества блокировок, а не количества пользователей. Важно учитывать, что переход к блокировке всей таблицы может произойти и после того, как пользователь приступит к осуществлению массового обновления. В таком случае блокировка, первоначально установленная на уровне строки, вначале заменяется блокировкой страницы, а затем в результате эскалации становится блокировкой таблицы. Это означает, что выполнение любой операции может в принципе послужить препятствием для работы с той же таблицей для всех прочих пользователей. А если в запросе используются многочисленные таблицы, то фактически возникает весьма значительная вероятность того, что все прочие пользователи не смогут работать со всеми этими таблицами.

Безусловно, такая организация работы, при которой действия, осуществляемые над определенным объектом одним пользователем, становятся препятствием для доступа к тому же объекту для всех прочих пользователей, вряд ли можно назвать приемлемой, но иногда действительно приходится выполнять такие операции обновления, которые по существу сводятся к этому эффекту. В действительности почти единственный способ, позволяющий предотвратить эскалацию блокировок, состоит в том, чтобы все применяемые запросы были как можно более целенаправленными. Необходимо всегда помнить о том, что возможна эскалация блокировок, поэтому стремиться учитывать все возможные последствия применения намеченных запросов.

Режимы блокировки

Продумывая организацию работы приложения, необходимо не только учитывать, на каком уровне должна происходить блокировка ресурсов, но и предусматривать применение определенного режима, с помощью которого должно происходить приобретение блокировок в запросе. А поскольку количество разнообразных ресурсов, которые могут быть заблокированы, достаточно велико, то велико и разнообразие **режимов блокировки**.

Некоторые режимы являются взаимно исключающими (это означает, что эти режимы не могут применяться в сочетании друг с другом). Но есть и такие режимы, которые применяются для модификации других режимов. Возможность совместного использования режимов зависит от того, являются ли они **совместимыми** (более подробное описание того, в чем состоит совместимость между блокировками, приведено ниже в этой главе).

В предыдущих разделах рассматривались блокируемые ресурсы, а в следующих разделах приведено последовательное описание режимов блокировки.

Разделяемые блокировки

Разделяемые блокировки относятся к наиболее простому типу блокировок. В частности, **разделяемые блокировки** используются, если требуется обеспечить только чтение данных, т.е. применительно к этим данным не планируется внесение каких-либо изменений. Чаще всего применение разделяемых блокировок не связано с какими-либо затруднениями, поскольку они совместимы с другими разделяемыми блокировками. Но из этого не следует, что при использовании блокировок этого типа не приходится ни о чем беспокоиться. Дело в том, что разделяемые блокировки могут применяться в сочетании почти со всеми прочими видами блокировок, но есть такие блокировки, которые исключают возможность применения разделяемых блокировок.

Применение разделяемых блокировок не приводит к созданию каких-либо особых ограничений. Тем не менее после ввода в действие разделяемой блокировки ее наличие должно учитываться при использовании блокировок других типов. Разделяемые блокировки применяются главным образом для того, чтобы была исключена возможность выполнять чтение незафиксированных данных пользователями.

Исключительные блокировки

Назначение **исключительных блокировок** полностью следует из их названия. Исключительные блокировки не могут применяться в сочетании с любыми другими блокировками. Если существуют какие-либо другие блокировки, приобретение исключительной блокировки становится невозможным. Кроме того, до тех пор пока все еще остается активной исключительная блокировка, не допускается создание новой блокировки любого типа на том же ресурсе. Благодаря этому исключается возможность для нескольких пользователей проводить одновременно операции обновления, удаления и тому подобные действия.

Блокировки обновления

Блокировки обновления представляют собой своего рода сочетание разделяемых и исключительных блокировок. Любую блокировку обновления можно рассматривать как метку-заполнитель особого типа. Назначение блокировок этого типа вполне очевидно — для того чтобы выполнить операцию обновления с помощью оператора `UPDATE`, необходимо вначале определить, какие строки подлежат обновлению с помощью конструкции `WHERE` (при условии, что таковые существуют). Из того, что вначале необходимо проверить наличие определенного количества строк, следует, что на первом этапе требуется только разделяемая блокировка, и такая ситуация сохраняется фактически до тех пор, пока не происходит переход к этапу физического обновления. А после наступления этого этапа требуется исключительная блокировка.

Блокировки обновления указывают, что имеется разделяемая блокировка, которая должна стать исключительной блокировкой после завершения первоначального просмотра данных, который осуществляется для определения того, какие именно данные должны быть обновлены. Таким образом, применение блокировок обновления основано на том факте, что любая операция обновления состоит из двух описанных ниже этапов.

- На первом этапе происходит считывание данных, соответствующих критериям конструкции `WHERE` (данных, подлежащих обновлению). Этот этап является неотъемлемой частью запроса на обновление, который осуществляется с помощью блокировки обновления.

- На втором этапе, если действительно решено выполнить обновление, разделяемая блокировка заменяется исключительной блокировкой. В противном случае блокировка обновления преобразуется в разделяемую блокировку.

Преимущество организации работы с использованием блокировок обновления состоит в том, что такие блокировки предотвращают возможность возникновения одного из типов **взаимоблокировок**. Взаимоблокировки представляют собой не разновидность блокировок, а ситуацию, в которой нормальное продолжение работы становится невозможным. В частности, взаимоблокировка возникает, если при использовании блокировок в одной из транзакций исключается вероятность успешного продолжения работы и освобождения ресурсов, поскольку эти ресурсы захвачены с помощью блокировок во второй транзакции, притом что во второй транзакции для освобождения захваченных ресурсов требуется доступ к ресурсам, которые удерживаются в первой транзакции.

Если не используются блокировки обновления, то вероятность возникновения подобных взаимоблокировок становится весьма значительной. В частности, рассмотрим такую ситуацию. Предположим, что с использованием разделяемых блокировок выполняются два запроса на обновление. В запросе A выборка завершена и наступил этап физического обновления. На этом этапе необходимо перейти к применению исключительной блокировки, но это невозможно, поскольку в запросе B еще не завершена выборка данных. Затем, после завершения выборки в запросе B, обнаруживается, что необходимо осуществить физическое обновление. Для этого в запросе B требуется перейти к использованию исключительной блокировки, но это невозможно, поскольку в запросе A еще не получен доступ к ресурсам, необходимый для эскалации блокировки до уровня исключительной. В результате возникает тупиковая ситуация.

С другой стороны, если в одной транзакции применяется блокировка обновления, то исключается возможность устанавливать во всех прочих транзакциях блокировки обновления для тех же ресурсов. С того момента, как в следующей транзакции предпринимается попытка приобрести блокировку обновления на заблокированных ресурсах, эта транзакция переводится в состояние ожидания на время, называемое *тайм-аутом блокировки*, но сама блокировка не предоставляется. Если первая транзакция освобождает блокировку до истечения тайм-аута блокировки во второй транзакции, то право установить блокировку передается тому, кто запросил ресурс во вторую очередь, и процесс обработки данных продолжается. В противном случае во второй транзакции возникает ситуация ошибки.

Блокировки обновления совместимы только с разделяемыми блокировками и намеченными разделяемыми блокировками (которые рассматриваются ниже).

Намеченные блокировки

Намеченная блокировка еще больше напоминает метку-заполнитель, чем блокировка обновления, и позволяет справиться с проблемами, которые возникают при использовании блокировок, способных охватывать разные уровни иерархии объектов. В частности, предположим, что в одной транзакции установлена блокировка на строке, а для выполнения второй транзакции требуется установить блокировку на странице. экстенте или даже модифицировать всю таблицу. Безусловно, необходимо исключить такую ситуацию, чтобы во второй транзакции был захвачен ресурс, на котором установлена блокировка в первой транзакции.

Если не используются намеченные блокировки, то объекты более высокого уровня могут оказаться полностью заблокированными в связи с тем, что уже установлена блокировка на более низком уровне. Применение намеченных блокировок способствует повышению производительности, поскольку в процессе эксплуатации СУБД SQL Server требуется проверять наличие намеченных блокировок только на уровне таблицы и не заниматься проверкой наличия блокировок на каждой строке или странице, относящейся к таблице, для определения того, можно ли в транзакции заблокировать всю таблицу, не опасаясь нарушить работу другой транзакции. Намеченные блокировки подразделяются на три типа, описанных ниже.

- **Намеченная разделяемая блокировка**. Разделяемая блокировка, которая установлена или должна быть установлена на каком-то объекте, находящемся на более низком уровне иерархии по сравнению с максимально допустимым. Например, такая блокировка применяется, если на странице необходимо установить разделяемую блокировку уровня страницы. Блокировки этого типа применяются только к таблицам и страницам.
- **Намеченная исключительная блокировка**. Блокировка этого типа аналогична намеченной разделяемой блокировке, но отличается тем, что в ней предусматривается установка исключительной блокировки на объекте более низкого уровня.
- **Разделяемая блокировка с намеченной исключительной блокировкой**. Разделяемая блокировка, которая установлена или должна быть установлена на объекте, находящемся в иерархии объектов на уровне ниже максимально допустимого, но с помощью этой блокировки намечено осуществление модификации данных, поэтому в какой-то момент она преобразуется в намеченную исключительную блокировку.

Блокировки схемы

Блокировки схемы подразделяются на два описанных ниже типа.

- **Блокировка модификации схемы**. Блокировки модификации схемы (Schema Modification lock — **Sch-M**) применяются для обеспечения возможности внесения изменений в схему объекта базы данных. До тех пор пока на объекте установлена блокировка Sch-M, к нему не будут применяться какие-либо запросы или операторы `CREATE`, `ALTER` и `DROP`, относящиеся к другим транзакциям.
- **Блокировка стабильности схемы**. Блокировки стабильности схемы (Schema stability lock — **Sch-S**) весьма напоминают разделяемые блокировки; их единственное назначение состоит в том, чтобы предотвратить создание блокировки Sch-M, поскольку на рассматриваемом объекте уже установлены блокировки, предназначенные для выполнения других запросов (или операторов `CREATE`, `ALTER` и `DROP`). Блокировки этого типа совместимы с блокировками всех других типов.

Блокировки массового обновления

Блокировки массового обновления (Bulk Update lock — **BU**) в действительности представляют собой одну из разновидностей блокировок таблицы, если не считать одного (но очень важного) различия. Блокировки массового обновления допускают параллельную загрузку данных; это означает, что при их использовании таблица бло-

кируется по отношению к любым прочим "обычным" операциям (выполняемым с помощью операторов T-SQL), но в то же время обеспечивается возможность выполнения многочисленных операций BULK INSERT или операций bcp.

Совместимость блокировок

Сведения о совместимости режимов блокировки ресурсов приведены в табл. 14.3 (блокировки показаны в порядке возрастания приоритета). Существующие блокировки упорядочены по столбцам, а запрашиваемые блокировки по строкам.

Таблица 14.3. Сведения о совместимости режимов блокировки ресурсов

	IS	S	U	IX	SIX	X
Намеченная разделяемая блокировка (IS)	Да	Да	Да	Да	Да	Нет
Разделяемая блокировка (S)	Да	Да	Да	Нет	Нет	Нет
Блокировка обновления (U)	Да	Да	Нет	Нет	Нет	Нет
Намеченная исключительная блокировка (IX)	Да	Нет	Нет	Да	Нет	Нет
Блокировка, разделяемая с намеченной исключительной (SIX)	Да	Нет	Нет	Нет	Нет	Нет
Исключительная блокировка (X)	Нет	Нет	Нет	Нет	Нет	Нет

Кроме того, необходимо учитывать описанные ниже условия применения блокировок.

- Блокировки Sch-S совместимы со всеми режимами блокировки, кроме Sch-M.
- Блокировки Sch-M несовместимы со всеми прочими режимами блокировки.
- Блокировки BU совместимы только с блокировками стабильности схемы (Sch-S) и другими блокировками массового обновления.

Определение конкретного типа блокировки с помощью подсказок оптимизатору

Иногда возникает необходимость в приобретении большего контроля над тем, каким образом осуществляется блокировка данных в ходе выполнения запроса или, возможно, даже всей транзакции. Этой цели можно достичь с использованием так называемых **подсказок оптимизатору**.

Подсказки оптимизатору предоставляют возможность явно указать СУБД SQL Server, что должна быть выполнена эскалация блокировки до определенного уровня. Текст подсказки оптимизатору должен находиться непосредственно после имени таблицы (в операторе SQL), на которую распространяется действие подсказки.

Необходимо отметить, что подсказки оптимизатору относятся к категории инструментария наиболее высокого уровня. Практика показывает, что эти программные средства используются неправильно даже опытными разработчиками приложений для SQL Server, поэтому не следует применять их без разбора.

Продумывая возможность применения подсказок оптимизатору, необходимо учитывать, что компанией Microsoft затрачены буквально миллионы долларов на усовершенствование таких программных компонентов, как оптимизатор запросов. На дан-

ный момент достигнуто такое положение, что практически невозможно найти лучшее решение по использованию тех или других блокировок, применяемых в различных ситуациях, по сравнению с решением, найденным оптимизатором. Подсказки оптимизатору, включенные в запрос, предназначены лишь для исправления решений, принятых оптимизатором, путем указания мелких подробностей, которые могут оказаться неизвестными оптимизатору. Но в подавляющем большинстве случаев никому не удается достичь и превзойти уровень знаний, которыми руководствовался коллектив, занимающийся разработкой оптимизатора. Поэтому следует прибегать к использованию подсказок оптимизатору, лишь пройдя все ступени изучения программных средств СУБД SQL Server (автор, со своей стороны, обещает описать эти средства более подробно в будущей версии данной книги, предназначенной для профессионалов).

Определение наличия блокировок с использованием программы Management Studio

По-видимому, наиболее удобный способ ознакомления с блокировками состоит в применении программы Management Studio. В программе Management Studio предусмотрена возможность просматривать с помощью окна Activity Monitor блокировки, отсортированные по двум разным признакам — по идентификатору процесса или по объектам.

Чтобы воспользоваться указанным окном с отображением блокировок программы Management Studio, перейдите к узлу Management⇨Activity Monitor, связанному с обозначением используемого сервера. Затем щелкните на указанном узле правой кнопкой мыши и выберите требуемый тип информации. После этого откроется новое окно, которое показано на рис. 14.3.

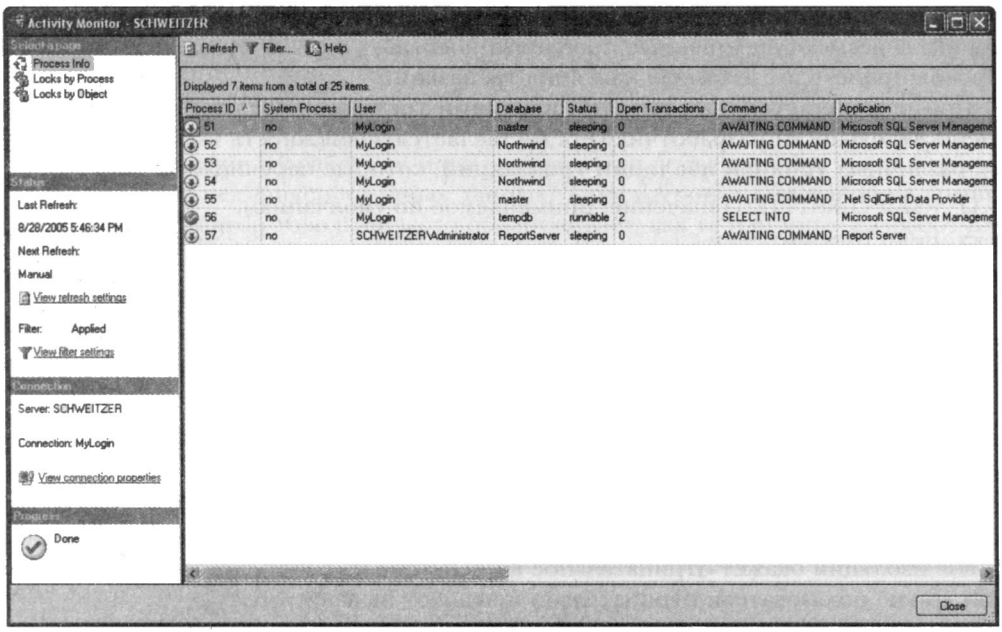


Рис. 14.3. Окно Activity Monitor

Для просмотра различных блокировок в этом окне достаточно развернуть интересующий вас узел (либо Lock by Process, либо Lock by Object).

Одно из наиболее привлекательных средств в этом окне обнаруживается после двойного щелчка кнопкой мыши на обозначении конкретной блокировки в правой части окна. Появляется диалоговое окно, в котором указан последний оператор, выполненный в процессе с данным идентификатором. Такая возможность становится очень удобной, если осуществляется поиск причин нарушений в работе, связанных с возникновением ситуаций взаимоблокировки.

Определение уровня изоляции транзакции

Выше в данной главе было описано, какие разнообразные проблемы могут быть устранены с использованием различных стратегий блокировки. Кроме того, описаны различные возможные типы блокировок и показано, как эти блокировки влияют на доступность ресурсов. А с этого раздела начинается подробное описание того, как взаимодействуют различные фрагменты указанных средств управления ресурсами в целях решения общей задачи обеспечения целостности данных — для гарантированного достижения ожидаемых результатов.

Чтобы понять, в чем состоит связь между транзакциями и блокировками, необходимо прежде всего выяснить, как выполняются транзакции в условиях применения блокировок. По умолчанию любая блокировка, применяемая для управления процессом модификации данных, после ее создания сохраняется в течение всей продолжительности выполнения транзакции. Это означает, что при использовании транзакций, осуществляемых в течение значительного времени, блокировки, которые предотвращают доступ других процессов к объектам, на которых они установлены, могут захватывать объекты на довольно длительное время. Очевидно, что из-за этого могут возникать весьма существенные проблемы, поскольку в связи с использованием блокировок происходит взаимная изоляция транзакций.

Тем не менее степень взаимной изоляции транзакций является управляемой. В действительности разработчику предоставляется возможность выбора среди четырех различных **уровней изоляции транзакций**. которые перечислены ниже.

- READ COMMITTED (значение, применяемое по умолчанию).
- READ UNCOMMITTED.
- REPEATABLE READ.
- SERIALIZABLE.

Для выбора конкретного уровня изоляции применяется довольно простой синтаксис:

```
SET TRANSACTION ISOLATION LEVEL <READ COMMITTED|READ UNCOMMITTED
   |REPEATABLE READ|SERIALIZABLE>
```

Изменение значения уровня изоляции отражается только на текущей транзакции, поэтому можно не задумываться над тем, что применение какого-то определенного уровня изоляции окажет отрицательное воздействие на работу других пользователей (или другие пользователи отрицательно повлияют на вашу работу).

Рассмотрим более подробно, какая ситуация возникает при использовании уровня изоляции, заданного по умолчанию (опция READ COMMITTED).

Опция READ COMMITTED

При использовании опции READ COMMITTED все созданные разделяемые блокировки автоматически освобождаются после завершения выполнения оператора, в котором они были созданы. Иными словами, в том случае, если начинается определенная транзакция, выполняется несколько операторов, после этого вызывается на выполнение оператор SELECT, а затем выполняется еще несколько операторов, то блокировки, связанные с оператором SELECT, освобождаются сразу после завершения его выполнения, т.е. СУБД SQL Server не ожидает завершения транзакции.

При выполнении запросов, которые оказывают воздействие на данные (с операторами UPDATE, DELETE или INSERT), возникает немного иная ситуация. Если в транзакции выполняется запрос, в котором модифицируются данные, то соответствующие блокировки сохраняются на время выполнения всей транзакции (на тот случай, если необходимо будет произвести откат).

Оставив неизменным уровень изоляции READ COMMITTED, предусмотренный по умолчанию, можно быть уверенным в том, что применяемые средства обеспечения целостности данных вполне позволят предотвратить чтение незафиксированных данных. Однако тем самым невозможно исключить вероятность неповторяемого чтения и появления фантомов.

Опция READ UNCOMMITTED

Опция READ UNCOMMITTED представляет собой наиболее опасный из всех вариантов выбора уровня изоляции, но обеспечивает наивысшую производительность, измеряемую скоростью выполнения транзакции.

Если уровень изоляции определен как READ UNCOMMITTED, это служит для СУБД SQL Server указанием на то, что не требуется устанавливать какие-либо блокировки, а также учитывать наличие каких-либо других блокировок. Если задан этот уровень изоляции, то пользователь может испытать на себе все возможные проблемы, связанные с нарушением правил параллельной организации работы, описанных выше в настоящей главе (причем наиболее важными из них является чтение незафиксированных данных).

По какой причине разработчик может согласиться подвергнуться риску, который связан с чтением незафиксированных данных? Просматривая обсуждения в группах новостей Usenet, автор довольно часто обнаруживает, как те или иные разработчики снова и снова возвращаются к данному вопросу. Для многих это остается непонятным, но существуют довольно весомые причины для использования указанного уровня изоляции, и почти всегда они связаны с необходимостью формирования отчетов.

В любой среде OLTP блокировки одновременно являются и вашим защитником, и вашим врагом. Блокировки позволяют предотвратить возникновение проблем нарушения целостности данных, но из-за них столь же часто предотвращается (или блокируется) возможность своевременного получения требуемых данных. Слишком часто приходится сталкиваться с такой ситуацией, когда руководители компании требуют регулярно предоставлять им отчеты, а сотрудники компании, занимающиеся вводом данных, не имеют возможности вводить информацию или испытывают задержки при вводе из-за блокировок, установленных в связи с формированием отчетов для руководителей.

Применение опции READ UNCOMMITTED часто позволяет обойти указанную проблему, по крайней мере при формировании отчетов, в которых не обязательно должны находиться абсолютно точные сведения. Например, предположим, что коммерче-

ский директор желает лишь узнать, какой объем продаж достигнут за сегодняшний день до настоящего времени. Безусловно, подобное поведение характерно для чрезмерно придирчивого директора, который задает один и тот же вопрос много раз в день (в форме повторного требования на получение одного и того же отчета), но мы обязаны выполнять это требование.

Если для формирования отчета требуется продолжительное время, то повышается вероятность того, что из-за блокировки ресурсов, связанной с подготовкой отчета, будет приостановлена работа других сотрудников. Но у любого подобного отчета есть замечательное свойство — он требуется лишь для приблизительной оценки ситуации, поэтому сбор абсолютно точных данных для его формирования, по-видимому, не имеет смысла. Директору в действительности требуются лишь оценочные данные.

Задавая для соответствующего запроса уровень изоляции `READ UNCOMMITTED`, мы не устанавливаем никаких блокировок, поэтому не препятствуем выполнению других транзакций. Безусловно, полученные при этом данные могут оказаться немного сомнительными (в связи с тем, что существует риск чтения незафиксированных данных), но так или иначе точные данные не требуются, к тому же известно, что итоговые значения все равно окажутся достаточно близкими к истинным, даже несмотря на наличие шансов, что произойдет откат и часть считанных данных будет относиться к незафиксированной транзакции.

Такого же эффекта, как и при использовании опции `READ UNCOMMITTED`, можно добиться, введя запрос в подсказку оптимизатору `NOLOCK`. Но подход, в котором используется опция, определяющая уровень изоляции, имеет свое преимущество в том, что не требуется задавать подсказку для каждой таблицы в запросе или использовать ее в нескольких запросах. С другой стороны, имеет свое преимущество и подход, в котором применяется подсказка оптимизатору `NOLOCK`, поскольку при этом не нужно помнить о том, что для данного конкретного соединения необходимо будет снова задать уровень изоляции, предусмотренный по умолчанию (а при использовании опции `READ UNCOMMITTED` такая необходимость возникает).

Опция REPEATABLE READ

Опция `REPEATABLE READ` обеспечивает определенную эскалацию уровня изоляции, что в значительной степени способствует повышению уровня защищенности параллельно выполняемых операций благодаря предотвращению чтения незафиксированных данных (эта задача решается и с помощью опции, предусмотренной по умолчанию), но вместе с тем исключает возможность неповторяемого чтения.

Безусловно, такая возможность предотвращения неповторяемого чтения является важным достижением, но реализация этого режима не всегда оправдана. Дело в том, что сохранение даже разделяемых блокировок до конца транзакции исключает возможность получения доступа к заблокированным объектам другими пользователями, поэтому отрицательно сказывается на производительности. Сам автор предпочитает использовать другие способы обеспечения целостности данных (такие как применение ограничения `CHECK` в сочетании со средствами обработки ошибок), а не этот вариант, но в определенных ситуациях применение этой опции вполне оправдано.

Уровню изоляции `REPEATABLE READ` соответствует эквивалентная подсказка оптимизатору `REPEATABLEREAD` (очевидно, что последнее ключевое слово отличается от первого лишь тем, что в нем отсутствует пробел).

Опция SERIALIZABLE

Опция SERIALIZABLE задает наиболее строгий из всех уровень изоляции. При ее использовании появляется возможность предотвратить возникновение всех форм нарушения правильной организации параллельной работы, кроме потерянных обновлений. Предотвращается даже возникновение фантомов.

Установка уровня изоляции, равного SERIALIZABLE, равносильна указанию, что любые операторы UPDATE, DELETE или INSERT, применяемые к той же таблице (или таблицам), которая используется в транзакции, не должны обращаться к тем же строкам, которые соответствуют условию WHERE любого оператора в этой транзакции. По существу, если любой другой пользователь собирается выполнить какие-либо действия над данными, представляющими интерес с точки зрения транзакции, для которой задана опция SERIALIZABLE, то он должен ожидать завершения этой транзакции.

Режим работы с уровнем изоляции SERIALIZABLE можно также моделировать, используя в запросе подсказку оптимизатору SERIALIZABLE или HOLDLOCK. Кроме того, как и при сравнении опции READ UNCOMMITTED и подсказки оптимизатору NOLOCK, необходимо учитывать, что опция SERIALIZABLE в отличие от подсказки оптимизатору SERIALIZABLE или HOLDLOCK должна быть указана в операторе только один раз, но, с другой стороны, при использовании подсказки оптимизатору не нужно помнить о том, что необходимо восстанавливать прежнее значение уровня изоляции.

На первый взгляд кажется, что лучше всего полностью перейти к использованию уровня изоляции SERIALIZABLE и отказаться от всех прочих опций. Безусловно, применение этой опции позволяет обеспечить достижение в процессе эксплуатации базы данных наивысшего уровня проявления замечательной характеристики данных, известной под названием ***совместимости****. Это означает, что операции модификации данных, одновременно выполняемые многочисленными пользователями, осуществляются так, как если бы в любой момент в базе данных выполнялась транзакция только одного пользователя (иными словами, обработка всех транзакций происходила бы последовательно).*

Тем не менее, как и за все хорошее в нашей жизни, за это также приходится платить. Практика показывает, что требования по обеспечению совместимости и достижению максимальной степени распараллеливания работы являются взаимоисключающими. Применение опции SERIALIZABLE может привести к тому, что другие пользователи не смогут получить доступ к тем объектам, которые для них требуются, а это равносильно уменьшению степени распараллеливания. Истинным является также противоположное утверждение – по мере увеличения степени распараллеливания (например, в результате перехода к использованию опции REPEATABLE READ) совместимость данных в базе данных уменьшается.

Личная рекомендация автора состоит в том, что следует придерживаться заданного по умолчанию значения (опции READ COMMITTED), если нет особых причин для отказа от этого.

Организация работы в условиях появления взаимоблокировок (при возникновении ошибки с номером 1205)

В предыдущих разделах приведены сведения о блокировках, а также описано, как блокировки применяются в транзакциях. На основании этих сведений мы можем перейти к рассмотрению довольно неприятной проблемы, связанной с возникновением **взаимоблокировок**.

Как уже было сказано, взаимоблокировки как таковые не относятся к категории блокировок. Чаще всего они характеризуются тем, что под воздействием применяемых блокировок возникает неразрешимая ситуация. С этим нарушением в работе рано или поздно приходится сталкиваться любому разработчику (особенно, если он не обладает достаточным опытом, чтобы обеспечить их предотвращение). О возникновении взаимоблокировки свидетельствует появление **ошибки с номером 1205**. Ситуация взаимоблокировки возникает настолько часто, что многие разработчики программного обеспечения для баз данных упоминают о ней просто по номеру.

Взаимоблокировки возникают, если невозможно завершить какое-либо действие, необходимое для освобождения одной из блокировок, поскольку требуемый для этого ресурс удерживается второй блокировкой, и наоборот. Если обнаруживается такая ситуация, то одна из операций должна быть завершена аварийно, для того чтобы вторая операция была доведена до конца, поэтому в СУБД SQL Server осуществляется выбор **жертвы взаимоблокировки**. После этого происходит откат транзакции, выбранной в качестве жертвы взаимоблокировки, и об этом поступает уведомление в виде сообщение об ошибке с номером 1205. После этого выполнение другой транзакции может быть продолжено обычным образом (безусловно, в самой транзакции наличие взаимоблокировки обнаружить невозможно, кроме как по возрастанию продолжительности выполнения).

Способы определения наличия взаимоблокировок в СУБД SQL Server

Через каждые пять секунд в СУБД SQL Server проверяются все текущие транзакции для определения того, освобождение каких блокировок требуется для их дальнейшего выполнения, тогда как сами эти транзакции еще не получили возможность установить данные блокировки. По существу, в ходе этой проверки регистрируется информация о том, какие существующие потребности в ресурсах еще не удовлетворены. А во время проведения следующей проверки состояния выполнения всех требований о предоставлении блокировок в СУБД предпринимается попытка выяснить, имеются ли еще не удовлетворенные требования, которые были обнаружены во время предыдущей проверки. В случае обнаружения таких требований осуществляется рекурсивная проверка всех открытых транзакций в целях обнаружения циклической цепочки требований на предоставление блокировок. Если такая цепочка обнаруживается, это означает, что имеет место взаимоблокировка; в таком случае одна из транзакций выбирается в качестве жертвы взаимоблокировки и осуществляется ее откат.

Принципы выбора жертвы взаимоблокировки

По умолчанию выбор жертвы взаимоблокировки осуществляется с учетом расхода ресурсов в каждой из затронутых транзакций. Операция отката применяется к той транзакции, в ходе выполнения которой был израсходован наименьший объем ресурсов (благодаря этому обеспечивается выполнение в СУБД SQL Server наименьшего объема работы при повторном выполнении всех операций восстанавливаемой транзакции). Разработчик имеет возможность в определенной степени повлиять на то, как осуществляется выбор жертвы взаимоблокировки, с использованием параметра `DEADLOCK_PRIORITY`, устанавливаемого в СУБД SQL Server с помощью оператора `SET`, но обычно не рекомендуется брать на себя такую ответственность, поэтому информация по данной теме выходит за рамки рассмотрения настоящей книги.

Предотвращение возникновения взаимоблокировок

Если система эксплуатируется достаточно интенсивно, то в ней невозможно полностью избежать возникновения взаимоблокировок, но на практике вполне осуществима задача почти полного их устранения. Иными словами, некоторые методы позволяют добиться того, чтобы взаимоблокировки почти не оказывали влияния на функционирование системы.

Ниже приведены простые правила (но иногда требующие значительных усилий), которые позволяют сократить или даже устранить взаимоблокировки.

- Во всех операциях доступ к объектам должен осуществляться в одном и том же порядке.
- Применяемые транзакции должны быть как можно более короткими, а операторы транзакций должны оформляться в виде одного пакета.
- Следует использовать наиболее низкий необходимый уровень изоляции транзакции из всех возможных.
- В той же транзакции, в которой осуществляется обработка данных, не следует допускать возникновения перерывов, время окончания которых не регламентируется (например, из-за взаимодействия с пользователем, перехода от одного пакета к другому и т.д.).
- В управляемой среде необходимо использовать связанные соединения (которые кратко описаны ниже).

Почти каждый раз, когда приходится сталкиваться с проблемами взаимоблокировки, обнаруживается нарушение по меньшей мере одного (а обычно даже большего количества) из этих правил. Ниже каждая из этих рекомендаций рассматривается более подробно.

Осуществление доступа к объектам во всех операциях в одном и том же порядке

Выше перечислены правила предотвращения взаимоблокировок, которые, по мнению автора, являются наиболее важными, но прежде всего необходимо обеспечить, чтобы доступ к объектам во всех операциях обработки данных всегда происходил в одном и том же порядке. Замечательная особенность этого правила заключается в том, что его соблюдение не требует почти никаких издержек; достаточно неизменно

придерживаться его в своей работе. На самых ранних этапах процесса проектирования необходимо принять решение о том, как должен осуществляться доступ к объектам базы данных, включая определение самой последовательности доступа, а после этого достаточно просто соблюдать привычный подход при написании кода каждого запроса, процедуры или триггера, создаваемого для данного проекта.

Суть приведенной рекомендации вполне очевидна — если проблема заключается в том, что с базой данных установлены два соединения и в каждом из этих соединений выдвигается требование на получение тех же ресурсов, что и в другом соединении, то нужно сделать вывод, что теперь уже поздно заниматься решением указанной проблемы. Рассмотрим простой пример.

Предположим, что в базе данных имеются две таблицы, `Suppliers` и `Products`. А теперь допустим, что в базе данных реализованы два процесса, в которых используются обе эти таблицы. В процессе 1 происходит прием информации, касающейся пополнения товарных запасов, данные таблицы `Products` обновляются с учетом поступления нового количества товаров, а затем обновления вносятся в таблицу `Suppliers` с учетом данных о суммарном количестве приобретенных товаров. С другой стороны, в процессе 2 регистрируется информация о сбыте; при этом вносится корректировка, затрагивающая суммарное количество проданных товаров, показанное в таблице `Suppliers`, после чего сокращение количества товарных запасов регистрируется в таблице `Products`.

Если оба эти процесса осуществляются параллельно, то нарушения в работе становятся неизбежными. В процессе 1 приобретается исключительная блокировка на таблице `Products`, а в процессе 2 исключительная блокировка устанавливается на таблице `Suppliers`. После этого в процессе 1 предпринимается попытка установить блокировку на таблице `Suppliers`, но процесс 1 вынужден перейти в состояние ожидания до того времени, как процесс 2 освободит свою существующую блокировку. Между тем в процессе 2 предпринимается попытка создать блокировку на таблице `Products`, но в нем приходится ожидать, пока процесс 1 освободит свою существующую блокировку. Возникает тупиковая ситуация, в которой оба процесса ожидают друг друга. После обнаружения такой ситуации в СУБД SQL Server приходится выбирать жертву взаимоблокировки.

Но в этом сценарии порядок действий можно изменить так, чтобы в процессе 2 вначале уменьшалось количество товарных запасов в таблице `Products` и только после этого обновлялось общее количество проданных товаров в таблице `Suppliers`. Такая организация работы функционально эквивалентна первому способу организации взаимодействия процессов, но для перехода к этой новой форме организации взаимодействия процессов не требуется никаких затрат. Тем не менее результат становится буквально поразительным, поскольку взаимоблокировки полностью исключаются (во всяком случае, между двумя этими процессами). Ниже приведен краткий обзор действий, происходящих при такой организации работы.

Если оба указанных процесса эксплуатируются одновременно, то в процессе 1 приобретается исключительная блокировка на таблице `Products` (до сих пор все остается неизменным). Затем в процессе 2 также предпринимается попытка установить блокировку на таблице `Products`, но этот процесс вынужден ожидать завершения работы с этой таблицей в процессе 1 (обратите внимание на то, что какие-либо действия с таблицей `Suppliers` еще не были выполнены). Процесс 1 завершает работу с таблицей `Products`, но не освобождает блокировку, поскольку транзакция еще не закончена.

Процесс 2 продолжает ожидать освобождения блокировки, установленной на таблице Products. На следующем этапе процесс 1 приступает к получению блокировки на таблице Suppliers. Процесс 2 продолжает ожидать освобождения блокировки, которая установлена на таблице Products. Процесс 1 завершает свою работу с таблицей Products и выполняет фиксацию или откат транзакции, в зависимости от обстоятельств, но в любом случае освобождает все блокировки. С этого времени процесс 2 получает возможность установить блокировку на таблице Products и осуществляет остальную часть транзакции без каких-либо дополнительных затруднений.

Итак, достаточно было изменить порядок доступа к таблицам в указанных двух запросах, и это позволило устранить потенциальную проблему возникновения взаимоблокировки. По возможности придерживайтесь этой рекомендации по соблюдению единой последовательности доступа к таблицам, что даст возможность намного успешнее справляться с взаимоблокировками.

Применение как можно более коротких транзакций, реализованных в виде одного пакета

Эта рекомендация относится к числу наиболее важных. Кроме того, у опытных разработчиков стремление к соблюдению такой рекомендации становится почти инстинктивным — они просто применяют ее на практике, не задумываясь над тем, что так действительно следует делать.

Мало того, если предыдущая рекомендация требовала осуществления хоть каких-либо действий, соблюдение данной рекомендации вообще не требует никаких усилий. Достаточно предусмотреть выполнение в транзакции лишь самых необходимых операций и исключить все остальное — нет ничего проще. Причины, по которым сокращение объема операций, выполняемых в транзакции, оказывает положительный эффект, являются вполне очевидными — чем больше времени занимает транзакция и чем больше объектов она затрагивает (в ходе выполнения предусмотренных в ней операций), тем выше вероятность того, что придется столкнуться с тем, что в каком-то другом процессе потребуется один или несколько из используемых в транзакции объектов (т.е. степень распараллеливания уменьшится). Если же транзакция остается короткой, то количество объектов, которые могут в принципе вызвать взаимоблокировку, сводится к минимуму, а также сокращается продолжительность времени, в течение которого приходится сохранять блокировки на этих объектах. В этом состоят основные соображения, лежащие в основе данной рекомендации.

Если же все операторы транзакции находятся в одном пакете, то сокращается количество циклов обмена данными сервером по сети в течение одной транзакции. В результате этого устраняются возможные задержки, из-за которых для завершения транзакции потребуется больше времени, а также освобождаются блокировки.

Использование наиболее низкого необходимого уровня изоляции транзакции из всех возможных

Эта рекомендация является далеко не такой простой, поэтому ее реализация требует определенной, достаточной серьезной проработки. В связи с этим не удивительно, насколько часто указанную рекомендацию просто не принимают во внимание. И в этом случае соблюдается известная в психологии аксиома: над тем, что действительно требует размышлений, чаще всего не размышляют вообще. Не будьте такими как все — уделите достаточное внимание этой проблеме.

В распоряжении разработчика имеется несколько различных уровней изоляции транзакции. По умолчанию предусмотрен уровень READ COMMITTED. Если же применяется более низкий уровень изоляции, то разделяемые блокировки сохраняются в течение более короткого промежутка времени, чем при использовании более высокого уровня изоляции, поэтому конкуренция за блокировки уменьшается.

Отказ от использования транзакций, время окончания которых не регламентируется

По-видимому, эта рекомендация в наибольшей степени соответствует здравому смыслу из всех прочих приведенных рекомендаций, но именно она игнорируется чаще всего из-за приверженности давно устаревшим принципам организации работы.

Разработчики с большим стажем (особенно те из них, которые работали в свое время с мэйнфреймами) чаще всего для предотвращения возникновения потерянных обновлений просто захватывали блокировку и удерживали ее до завершения работы с таблицей. Но невозможно даже описать, какие проблемы возникали в связи с применением такой организации работы.

Рассмотрим следующий сценарий (который взят из реальной жизни). Сотрудникам коммерческого отдела предоставлена возможность использовать формы двух типов: предназначенные для обновления (с исключительными блокировками) и предназначенные для просмотра данных (с разделяемыми блокировками). Один из сотрудников указанного отдела обязан контролировать ход работ, но для этого он обычно использует формы, предназначенные для обновления. Он открывает такую форму и, не закрыв ее, отправляется на обед, продолжающийся достаточно долго (1–2 часа). Но в форме обновления остается открытой строка, которая теперь заблокирована на все время отсутствия этого сотрудника.

Но ситуация может развиваться по еще более худшему сценарию. В эпоху мэйнфреймов гораздо чаще, чем сейчас, применялась концепция организации очередей (фактически структуры данных на основе очередей могут оказаться весьма эффективными). Теперь предположим, что какой-то другой сотрудник запускает задание по выводу на печать (которое ставится в очередь) для вывода отчета, включающего информацию о том заказе, который открыл у себя в форме сотрудник, прежде чем отправиться на обед. Задание должно оставаться в очереди до тех пор, пока не будет снята блокировка с соответствующей строки. А поскольку очередь печати обрабатывается последовательно, то все задания на печать, введенные в компании, накапливаются вслед за этим первым заданием (и работа всех сотрудников останавливается из-за того, что один человек отправился на обед, не освободив перед этим одну-единственную строку).

Безусловно, этот пример показывает крайне неблагоприятное развитие ситуации, но вполне очевидно, что он позволяет наглядно подчеркнуть мысль, высказанную в этом разделе. Не следует ни в коем случае создавать блокировки, которые сохраняются на все время выполнения того или иного процесса, сроки завершения которого не регламентированы. Обычно под этим подразумевается взаимодействие с пользователем (как в приведенном выше примере с сотрудником, отправившимся на обед), но фактически аналогичную роль может играть любой процесс, для которого не определено время окончания.

Резюме

Транзакции и блокировки лежат в основе функционирования СУБД SQL Server, поэтому успешная разработка любых приложений для СУБД SQL Server неосуществима без правильного использования этих средств обеспечения целостности данных.

Транзакции позволяют обеспечить реализацию любых совокупностей операций в виде отдельных единиц работы, либо выполняемых целиком, либо не выполняемых вообще. А применение блокировок в СУБД SQL Server гарантирует, что в максимально возможной степени удастся избежать возникновения нарушений в работе, связанных с одновременным осуществлением в базе данных многочисленных операций (безусловно, полностью исключить возникновение таких нарушений в работе почти невозможно, но благодаря продуманной организации функционирования удается добиться очень многого). В результате правильного использования транзакций и блокировок в определенных сочетаниях база данных приобретает так называемые **свойства ACID** (Atomicity, Consistency, Isolation, Durability – неразрывность, согласованность, изолированность, устойчивость). Краткие определения свойств ACID приведены ниже.

- **Неразрывность**. Транзакция выполняется по принципу "все или ничего".
- **Согласованность**. Все ограничения и другие правила обеспечения целостности данных строго соблюдаются, и полностью обновляются все взаимосвязанные объекты (страницы данных, страницы индексов).
- **Изолированность**. Каждая транзакция полностью изолирована от всех прочих транзакций. На действия, осуществляемые в одной транзакции, невозможно оказать влияние, предусматривая выполнение каких-либо других действий в отдельной транзакции.
- **Устойчивость**. После завершения транзакции ее результаты передаются в систему на постоянное хранение. После этого данные находятся в безопасности, в том смысле, что из-за прекращения подачи электроэнергии или другого нарушения в работе системы, не связанного с отказом жесткого диска, не происходит полная или частичная потеря данных.

Таким образом, применение транзакций и блокировок позволяет свести к минимуму вероятность возникновения взаимоблокировок, гарантирует целостность данных и позволяет повысить общую производительность системы.

В следующей главе рассматриваются триггеры и показано, что они широко используются для реализации транзакций и осуществления операций отката.

15

Триггеры

Триггеры относятся к числу таких программных средств SQL, без которых иногда очень трудно обойтись, но в любом случае применение триггеров должно быть тщательно обосновано.

В настоящей главе приведен достаточный объем информации для того, чтобы можно было приступить к практическому использованию триггеров.

Кроме того, в этой главе рассматриваются основные аспекты применения триггеров. Ниже перечислены основные представленные здесь темы.

- Общее определение понятия триггера.
- Использование триггеров для создания более разносторонних средств обеспечения ссылочной целостности.
- Использование триггеров для создания мощных правил проверки целостности данных.
- Использование триггеров `INSTEAD OF` для создания более сложных обновляемых представлений.
- Другие общие способы использования триггеров.
- Управление последовательностью запуска триггеров.
- Проблемы повышения производительности.

В целом изучение сведений, представленных в данной главе, позволяет понять, насколько сложными являются решения о том, где и при каких обстоятельствах следует и не следует использовать триггеры. Кроме того, в этой главе показано, насколько мощными и разносторонними являются указанные программные конструкции.

Автор главным образом стремился достичь того, чтобы у читателя не сложилось впечатление, будто использование триггеров связано со слишком большими сложностями, поэтому следует избегать их всеми способами (хотя такое мнение разделяют очень многие специалисты по СУБД SQL Server). С другой стороны, не следует вдаваться в другую крайность и рассматривать триггеры как основу решения всех про-

блем организации работы базы данных. Лучше всего придерживаться такого подхода, что триггеры позволяют сделать очень многое, но могут также создать и достаточное количество проблем. Поэтому основой успеха является применение триггеров там, где они позволяют успешно справиться с работой, и отказ от них в тех случаях, если это нецелесообразно.

Ниже перечислены некоторые наиболее характерные области применения триггеров.

- Обеспечение ссылочной целостности. Безусловно, при любой возможности рекомендуется использовать декларативные средства обеспечения ссылочной целостности (Declarative Referential Integrity – DRI), но во многих случаях средства DRI не позволяют справиться с работой (например, с их помощью невозможно обеспечить поддержку ссылочной целостности с охватом нескольких баз данных и тем более серверов, они не поддерживают многие сложные типы связей и т.д.).
- Создание контрольных журналов. Под этим подразумевается ведение хронологии, позволяющей отслеживать состояние не только наиболее актуальных данных, но и действительную историю модификации каждой строки.
- Поддержка функциональных средств, подобных ограничению `CHECK`. В отличие от ограничения `CHECK` функциональные возможности триггеров распространяются на группы таблиц, баз данных и даже серверов.
- Подстановка других операторов вместо операторов модификации данных, применяемых пользователем. Обычно это направление использования триггеров предназначено для обеспечения вставки данных в сложные представления.

Кроме того, предусмотрена возможность применения триггеров нового типа (триггеров DDL), которые позволяют следить за изменениями в структуре таблиц. По-видимому, с этой необходимостью приходится сталкиваться достаточно редко (как уже было сказано, триггеры DDL – это триггеры нового типа, поэтому требуется определенное время для проверки целесообразности их использования).

Перечисленные области применения составляют лишь небольшую часть тех задач, основой решения которых могут служить триггеры. В следующем разделе приведено определение понятия триггера.

Общее определение понятия триггера

Триггер – это хранимая процедура особого типа, вызываемая на выполнение в ответ на определенные события. Триггеры подразделяются на два основных типа: триггеры языка определения данных (Data Definition Language – DDL) и триггеры языка манипулирования данными (Data Manipulation Language – DML).

Триггеры DDL активизируются в ответ на внесение каких-либо изменений в структуру базы данных теми или другими пользователями (под этим подразумеваются изменения, осуществляемые с помощью операторов `CREATE`, `ALTER`, `DROP` и т.п.). Триггеры этого типа были впервые введены в версии SQL Server 2005. Без триггеров DDL очень трудно обойтись в некоторых инсталляциях (особенно если инсталлированные программные средства должны обеспечивать высокую степень защиты), но в целом они имеют довольно узкую область применения. В общем, чаще всего приходится сталкиваться с тем, что триггеры DDL используются, если есть необходимость чрезвычайно строго следить за внесением изменений в структуру базы данных

и регистрировать хронологию этих изменений. В основе применения этих триггеров лежат чрезвычайно сложные методы, поэтому они упоминаются в настоящей книге лишь для предоставления информации о том, что такие конструкции существуют, и в связи с этим в настоящей главе речь идет только о более широко применяемых на практике триггерах DML.

Триггеры DML представляют собой фрагменты кода, которые закрепляются за конкретной таблицей или представлением. В отличие от хранимых процедур, при использовании которых необходимо явно вызывать на выполнение определенный код, триггеры вызываются на выполнение автоматически при обнаружении события (событий), связанного с выполнением операций над таблицей, за которой закреплен триггер. Безусловно, триггеры не могут быть вызваны явно; единственный способ обеспечения такого вызова состоит в выполнении требуемого действия над таблицей, за которой закреплен триггер.

Различия между хранимыми процедурами и триггерами не ограничиваются тем, что невозможен явный вызов триггера. Хранимые процедуры не только вызываются явно, но и отличаются двумя особенностями, которыми не обладают триггеры: принимают параметры и возвращают коды завершения.

Триггеры не имеют параметров, но в коде триггера может использоваться определенный механизм, позволяющий выяснить, на какие строки должно распространяться действие триггера (дополнительная информация по этой теме приведена ниже в данной главе). Еще одна особенность триггеров состоит в том, что в них допускается применение ключевого слова RETURN, но они не позволяют возвращать конкретные значения кода завершения (дело в том, что явный вызов триггера не предусмотрен, поэтому не определена точка вызова, в которую можно было бы возвратить код завершения).

Возможность закрепления триггеров за конкретными операторами определяется тем, что в языке SQL предусмотрены три типа запросов, предназначенных для внесения изменений в данные. В связи с этим предусмотрены три основных типа триггеров, а также дополнительные типы, создаваемые с учетом одновременного возникновения и совпадения событий, обусловленных выполнением запросов этих типов. Кроме того, выбор типа триггера зависит от того, как связаны во времени запуск триггера и возникновение соответствующих событий. Основные типы триггеров перечислены ниже.

1. Триггеры `INSERT`.
2. Триггеры `DELETE`.
3. Триггеры `UPDATE`.
4. Триггеры, создаваемые с учетом одновременного возникновения и совпадения событий.

Будет ли произведен запуск триггера (т.е. вызов на выполнение кода триггера), зависит не только от того, осуществляется ли действие, относящееся к той категории, на которую должен реагировать триггер. Еще одно условие запуска триггера определяется тем, относится ли операция, выполняемая в базе данных, к категории действий, регистрируемых в журнале, или нет. Например, действие, связанное с выполнением оператора DELETE, представляет собой обычное регистрируемое в журнале действие, при обнаружении которого происходит запуск всех соответствующих триггеров DELETE, а оператор TRUNCATE TABLE, выполнение которого также приводит к удалению строк, вызывает лишь освобождение пространства, которое использовалось для данных таблицы. Тем не менее при его выполнении не происходит удаление отдельно каждой строки, поэтому триггер не активизируется.

Синтаксис операторов создания триггеров во многом напоминает синтаксис всех прочих операторов CREATE, за одним исключением — в этом операторе должна быть указана таблица, за которой закрепляется триггер, поскольку триггер не может применяться отдельно от таблицы.

Синтаксис оператора CREATE TRIGGER является следующим:

```
CREATE TRIGGER <trigger name>
   ON [<schema name>.]<table or view name>
   [WITH ENCRYPTION | EXECUTE AS <CALLER | SELF | <user> >]
   {{{FOR|AFTER} <[DELETE] [,] [INSERT] [,] [UPDATE]>} |INSTEAD OF}
   [WITH APPEND]
   [NOT FOR REPLICATION]
AS
   < <sql statements> | EXTERNAL NAME <assembly method specifier> >
```

Вполне очевидно, что оператор создания триггера имеет знакомую структуру CREATE <object type> <object name>, а также включает в себя такое же определение исполняемого кода, как и во многих других объектах. В этом операторе введена лишь дополнительная конструкция ON, позволяющая указать таблицу, за которой должен быть закреплен триггер, а также задано, когда и при каких условиях должен происходить запуск триггера.

Конструкция ON

Конструкция ON оператора создания триггера предназначена лишь для указания объекта, к которому применяется создаваемый триггер. Но необходимо учитывать, что если триггер относится к типу AFTER (т.е. для объявления триггера используется ключевое слово FOR или AFTER), то назначением конструкции ON должна быть таблица, поскольку триггеры AFTER не предназначены для применения с представлениями.

Ключевое слово WITH ENCRYPTION

Ключевое слово WITH ENCRYPTION выполняет такие же функции, как и в операторах создания представлений и хранимых процедур. Если указана эта опция, то можно не сомневаться в том, что больше никто не увидит в базе данных исходный код соответствующего программного объекта (даже тот, кто сам его создал!). Такая возможность становится особенно удобной, если разрабатываемое программное обеспечение должно войти в состав коммерческого дистрибутива или к защите предъявляются строгие требования, согласно которым пользователи не должны знать, какие данные модифицируются и как происходит доступ к данным. Безусловно, в таком случае копия кода, требуемая для создания триггера, должна остаться где-то в другом месте, на тот случай, если когда-либо в дальнейшем потребуется снова его создать.

Как и в случае представлений и хранимых процедур, при использовании опции WITH ENCRYPTION следует помнить, что эту опцию необходимо задавать повторно при любом внесении изменений в код триггера с помощью оператора ALTER. Если код триггера модифицируется с помощью оператора ALTER TRIGGER, но при этом не используется опция WITH ENCRYPTION, то в своем новом варианте триггер больше не будет зашифрован.

Преимущества и недостатки конструкций FOR (AFTER) и INSTEAD OF

При определении триггера предусмотрена возможность не только указать, в связи с выполнением каких запросов (с операторами INSERT, UPDATE и (или) DELETE) должен происходить запуск триггера, но и, в определенной степени, определить, как должны быть связаны во времени выполнение запроса и запуск триггера. Безусловно, наиболее продолжительную историю применения имеют триггеры FOR (вместо этого ключевого слова, при желании, можно воспользоваться при объявлении триггера ключевым словом AFTER), поэтому многие разработчики привыкли применять триггеры этого типа, но предусмотрена также возможность эксплуатировать так называемые триггеры INSTEAD OF. От выбора триггера того или другого типа зависит, будет ли существовать возможность ввести в действие код триггера до или после модификации данных. Но в любом случае код триггера вызывается на выполнение прежде, чем любые внесенные изменения действительно будут зафиксированы в базе данных.

Приведенное здесь описание можно проще всего понять с помощью схемы, на которой показано, как происходит запуск триггеров FOR (AFTER) и INSTEAD OF (рис. 15.1).

Но важно отметить, что независимо от используемого типа триггера для реализации действий, предусмотренных этим триггером в СУБД SQL Server, используются две рабочие таблицы. В одной из этих таблиц хранятся копии всех вставляемых строк (в связи с этим данная таблица именуется INSERTED), а во второй хранятся копии всех удаляемых строк (эта таблица именуется DELETED). Подробные сведения о том, как используются в СУБД эти рабочие таблицы, приведены ниже. А на данный момент достаточно отметить, что при выполнении действий, предусмотренных в коде триггера INSTEAD OF, создание этих рабочих таблиц происходит до проверки каких-либо ограничений, а при обслуживании триггера FOR эти таблицы создаются после проверки ограничений. Ключевой особенностью триггера INSTEAD OF является то, что он фактически предназначен для выполнения кода, предусмотренного программистом, вместо того кода, выполнение которого обусловлено запросом, поступившим от пользователя. Это означает, что с помощью триггера INSTEAD OF могут быть устранены проблемы неоднозначности при вставке данных в представления (напомним, что в главе 10 был приведен пример, в котором проблема возникала при вставке данных в представление, основанное на использовании оператора JOIN). Это также означает, что с помощью триггера INSTEAD OF могут быть предприняты действия по предотвращению возможных нарушений тех требований, которые обусловлены в ограничениях, даже до того, как произойдет проверка этих ограничений.

Безусловно, такие возможности триггеров выглядят весьма привлекательно, но фактически задача реализации этих возможностей остается довольно сложной. Прежде всего, разрабатывая код триггера, необходимо учитывать все возможные варианты развития событий. Кроме того, в связи с применением триггера фактически должна быть предусмотрена предобработка перед каждым запросом, в котором так или иначе модифицируются данные таблицы (а это может отрицательно повлиять на производительность). Притом что триггеры INSTEAD OF предоставляют великолепные возможности, они относятся к категории очень сложных программных средств, поэтому их описание в основном выходит за рамки настоящей книги.

Триггеры, в объявлении которых используется ключевое слово FOR или AFTER, действуют одинаково. Но триггеры этого типа отличаются от триггеров INSTEAD OF тем, что для них рабочие таблицы создаются после проверки ограничений.

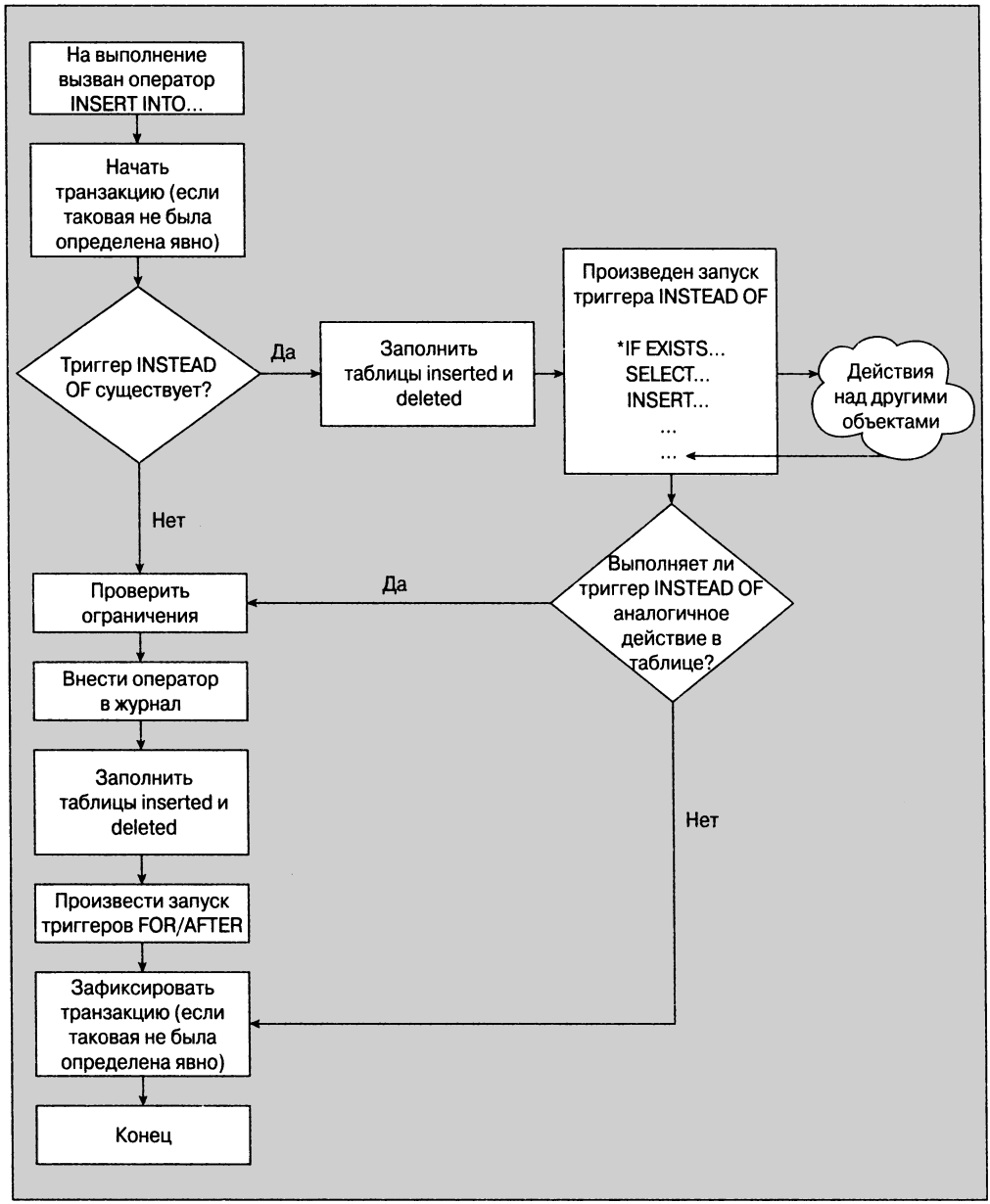


Рис. 15.1. Схема запуска триггеров FOR (AFTER) и INSTEAD OF

Конструкция FOR (AFTER)

Конструкция FOR (вместо нее вполне допустимо применять конструкцию AFTER) позволяет указать, какое действие (действия) приводит к запуску триггера. Может быть предусмотрен запуск триггера при выполнении оператора INSERT, UPDATE или DELETE либо любого сочетания этих трех операторов. В качестве примера ниже приведено несколько различных вариантов конструкции FOR.

```
FOR INSERT, DELETE
```

или

```
FOR UPDATE, INSERT
```

или

```
FOR DELETE
```

Как было указано в предыдущем разделе, посвященном описанию конструкции ON, триггеры, объявляемые с конструкцией FOR или AFTER, могут быть закреплены только за таблицами; закрепление этих триггеров за представлениями не допускается (дополнительная информация по этой теме приведена при описании триггеров INSTEAD OF).

Триггеры INSERT

Код любого триггера, который объявлен с ключевыми словами FOR INSERT, вызывается на выполнение каждый раз, когда кто-либо вставляет новую строку в таблицу, за которой закреплен триггер. Для каждой вставляемой строки СУБД SQL Server создает копию этой новой строки и вставляет ее в специальную таблицу, существующую только в области определения данного триггера. Эта таблица называется INSERTED, и дополнительные сведения о ней будут приведены в ходе описания различных тем настоящей главы. Наиболее важная особенность таблицы INSERTED состоит в том, что она сохраняется только до тех пор, пока действует триггер, с которым она связана. Следует еще раз подчеркнуть, что таблица INSERTED существует только с момента запуска триггера и до момента завершения его выполнения.

Триггеры DELETE

Триггеры DELETE действуют во многом аналогично триггерам INSERT, не считая того, что связанная с ними таблица INSERTED пуста (и это очевидно, поскольку в триггерах DELETE осуществляется удаление, а не вставка строк, поэтому отсутствуют строки, копия которых должна находиться в таблице INSERTED). Вместо этого копия каждой удаленной строки вставляется в другую таблицу, называемую DELETED. Эта таблица, как и таблица INSERTED, имеет область определения, которая ограничивается исключительно продолжительностью существования триггера.

Триггеры UPDATE

Триггеры UPDATE имеют свою специфику. Запуск кода триггера, объявленного с ключевыми словами FOR UPDATE, происходит при каждом внесении изменения в строку, существующую в таблице. А особенность триггера UPDATE состоит в том, что отсутствует такая таблица, как UPDATED. Вместо этого в СУБД SQL Server операция модификации каждой строки трактуется так, как будто удаляется существующая строка и вставляется полностью новая. Вполне очевидно, что из этого следует такой вывод: триггер, объявленный с ключевыми словами FOR UPDATE, обслуживается с по-

мощью не одной, а двух специальных таблиц, INSERTED и DELETED. Безусловно, в процессе работы количество строк в этих двух таблицах полностью совпадает.

Ключевое слово WITH APPEND

Ключевое слово WITH APPEND предназначено для устранения некоторых ограничений предыдущих версий SQL Server, поэтому после перехода на использование более современных версий необходимость в его применении фактически отпадает. Тем не менее в настоящем разделе это ключевое слово кратко описано на тот случай, что читателю придется с ним столкнуться. Ключевое слово WITH APPEND используется, только если СУБД эксплуатируется в режиме совместимости с версией 6.5 (такой режим может быть задан с помощью процедуры sp_dbcmptlevel).

В версии SQL Server 6.5 и предыдущих версиях не была предусмотрена возможность использования нескольких триггеров одного и того же типа применительно к какой-либо отдельной таблице. Например, если уже был объявлен триггер trgCheck, предназначенный для обеспечения целостности данных при выполнении операций обновления и вставки, то исключалась возможность создать отдельный триггер для каскадных обновлений. После создания единственного триггера для обновления (или вставки, или удаления) на этом все заканчивалось, поскольку невозможно было предусмотреть использование еще одного триггера для выполнения действия такого же типа.

В связи с этим возникали действительно серьезные затруднения, которые были обусловлены тем, что в одном триггере приходилось комбинировать логически разные действия. В связи с этим иногда было весьма нелегко обеспечить совместное применение в одном фрагменте кода двух полностью различных процедур. Кроме того, затруднялось также чтение создаваемого при этом весьма запутанного кода.

Наконец, была выпущена версия SQL Server 7.0, после чего правила применения триггеров существенно изменились. При использовании этой версии больше не приходилось задумываться над тем, какое количество триггеров разрешается использовать по отношению к запросу на выполнение действия одного типа; при желании можно было применять несколько таких триггеров. Тем не менее при эксплуатации базы данных в режиме совместимости с версией 6.5 возникали проблемы, поскольку база данных по-прежнему эксплуатировалась на основании того требования, что применительно к каждой конкретной таблице должен быть задан только один триггер определенного типа.

Ключевое слово WITH APPEND позволяет обойти такую проблему, поскольку служит для СУБД SQL Server указанием на то, что этот новый триггер должен быть введен в действие, даже несмотря на то, что на таблице уже определен триггер такого же типа. В связи с этим после осуществления соответствующего триггерного действия (выполнения оператора INSERT, UPDATE или DELETE) происходит запуск и старого, и нового триггеров. Благодаря этому появляется возможность использовать сочетание средств старой и новой версий.

Ключевое слово NOT FOR REPLICATION

Введение в действие опции NOT FOR REPLICATION приводит к небольшому изменению правил, в соответствии с которыми определяется время запуска триггера. Если эта опция введена в действие, то триггер не запускается каждый раз, когда модификация таблицы происходит в связи с выполнением задачи, относящейся к репликации. Дело в том, что обычно триггер запускается (для выполнения служебных действий,

обеспечения каскадного распространения операций и т.д.) во время модификации оригинала таблицы, поэтому нет смысла снова запускать его применительно к реплицируемой копии таблицы.

Ключевое слово AS

Так же как и в объявлениях хранимых процедур, за ключевым словом AS следует наиболее важная часть оператора. С помощью ключевого слова AS в СУБД SQL Server передаются сведения о том, запуск какого кода должен быть осуществлен при вызове триггера. Дальнейшее описание в настоящей главе посвящено только этой части триггера, фактически представляющей собой сценарий.

Использование триггеров для реализации правил обеспечения целостности данных

Прежде всего, триггеры могут использоваться для осуществления таких же функциональных возможностей, которые реализуются с помощью ограничений CHECK или даже конструкций DEFAULT, хотя при этом следует учитывать, что такой вариант применения триггеров не является наиболее приемлемым. На вопрос о том, следует ли применять триггеры или ограничения CHECK, можно дать лишь один определенный ответ – все зависит от обстоятельств. Если стоящая перед вами задача может быть выполнена с помощью ограничения CHECK, то, по-видимому, вариант, предусматривающий его использование, – наиболее предпочтительный. Но иногда ограничение CHECK просто не позволяет достичь намеченной цели, или в связи с применением этого ограничения приходится сталкиваться с такими затруднениями, что этот вариант становится менее приемлемым по сравнению с триггером. Ниже приведены примеры ситуаций, в которых вариант, основанный на использовании триггера, является более удобным по сравнению с ограничением CHECK.

- Применяются такие бизнес-правила, которые требуют обращения к данным еще в одной, отдельной таблице.
- Согласно используемому бизнес-правилу, должна проводиться проверка **дельты обновления** (различия между предыдущим и последующим состояниями данных).
- Должно быть предусмотрено формирование сообщений об ошибках, определяемых пользователем.

Итоговая таблица с рекомендациями по использованию различных механизмов обеспечения целостности данных приведена в конце главы 6 (см. табл. 6.4).

Приведенные выше примеры фактически демонстрируют лишь малую часть возможных областей применения триггеров. Безусловно, триггеры предоставляют чрезвычайно широкие возможности, поэтому в действительности решение об их использовании может быть принято лишь с учетом того, какие особые операции должны осуществляться в процессе эксплуатации базы данных.

Учет требований, связанных с совместным использованием нескольких таблиц

Преимущество ограничений CHECK заключается в том, что эти программные средства являются быстродействующими и эффективными, но, к сожалению, с их помощью нельзя реализовать все необходимые действия по обеспечению целостности данных. По-видимому, наиболее существенный недостаток ограничений CHECK обнаруживается после того, как возникает необходимость выполнить проверку данных с охватом нескольких таблиц.

В качестве иллюстрации рассмотрим таблицы Products и Order Details базы данных Northwind. Связь между этими таблицами показана на рис. 15.2.

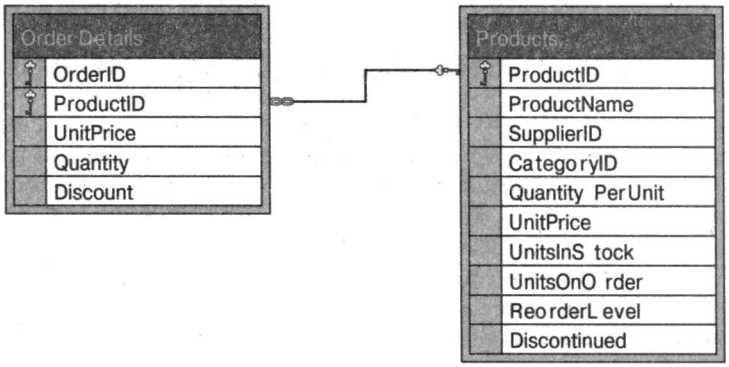


Рис. 15.2. Связь между таблицами ***Products*** *и* ***Order Details***

Таким образом, если используются обычные средства обеспечения ссылочной целостности, то можно быть уверенным в том, что в таблицу Order Details невозможно будет вставить ни одного элемента расшифровки заказа с наименованием товара, если в таблице Products отсутствуют данные о товаре с соответствующим ему идентификатором ProductID. Однако для решения сформулированной выше задачи требуется нечто большее, чем "обычные" средства.

Специалисты из отдела снабжения сообщают, что из торгового отдела по-прежнему поступают заказы на товары, прием которых на склад прекращен. В связи с этим высказано пожелание, чтобы попытки ввода подобных заказов отвергались еще до того, как эти заказы попадут в систему.

Указанную задачу невозможно решить с помощью ограничения CHECK, поскольку о том, что поставка товара прекращена, можно узнать только с помощью отдельной таблицы (таблицы Products), на основании данных которой должно контролироваться ограничение (которое распространяется на таблицу Order Details). Но такую задачу несложно решить с помощью триггера:

```
CREATE TRIGGER OrderDetailNotDiscontinued
   ON [Order Details]
   FOR INSERT, UPDATE
AS
   IF EXISTS
      (
       SELECT 'True'
```

```
    FROM Inserted i
    JOIN Products p
      ON i.ProductID = p.ProductID
    WHERE p.Discontinued = 1
   )
BEGIN
   RAISERROR('Order Item is discontinued. Transaction Failed.',16,1)
   ROLLBACK TRAN
END
```

Проверим полученные результаты. Прежде всего необходимо убедиться в том, что в таблице Products имеется одна или несколько строк, при обнаружении которых будет происходить запуск триггера:

```
SELECT ProductID, ProductName FROM Products WHERE Discontinued = 1
ProductID                   ProductName
----------------            -----------------------------------
5                           Chef Anton's Gumbo Mix
9                           Mishi Kobe Niku
17                          Alice Mutton
24                          Guarana Fantastica
28                          Rossle Sauerkraut
29                          Thuringer Rostbratwurst
42                          Singaporean Hokkien Fried Mee
53                          Perth Pasties
(8 row(s) affected)
```

Итак, продолжим проверку и введем элемент Order Details, который не соответствует заданному ограничению:

```
INSERT [Order Details]
   (OrderID, ProductID, UnitPrice, Quantity, Discount)
VALUES
   (10000, 5, 21.35, 5, 0)
```

В результате попытка ввода недопустимых данных будет отвергнута, как и следовало ожидать:

```
Msg 50000, Level 16, State 1, Line -1074284106
Order Item is discontinued. Transaction Failed.
```

Необходимо также учитывать, что можно было бы также при желании создать определяемое пользователем сообщение об ошибке, которое активизировалось бы вместо произвольного сообщения, используемого в команде RAISERROR.

Применение триггеров для проверки дельты обновления

Иногда интерес представляет не то, каким было или стало значение в результате обновления, а то, насколько изменилось это значение. Безусловно, подобную информацию невозможно найти в каком-либо одном столбце или таблице, но величину изменения можно вычислить с помощью триггера, воспользовавшись таблицами Inserted и Deleted.

Чтобы ознакомиться с этой возможностью, еще раз рассмотрим таблицу Products. В таблице Products имеется столбец UnitsInStock. Недавно произошло значительное повышение спроса на некоторые товары, и компания Northwind почти полнос-

тью распродала все товары некоторых наименований. Но руководство компании Northwind стремится к тому, чтобы в долговременном плане смогли продержаться на плаву все заказчики, а не только некоторые из них (наиболее энергичные), поэтому было решено ввести в действие систему нормирования товаров, пользующихся высоким спросом. Из отдела снабжения поступило требование, согласно которому запрещается принимать заказы, для выполнения которых потребуется отпустить больше половины запаса какого-то конкретного товара.

Для решения этой задачи можно воспользоваться одновременно таблицами `Inserted` и `Deleted`:

```
CREATE TRIGGER ProductIsRationed
   ON Products
   FOR UPDATE
AS
   IF EXISTS
      (
       SELECT 'True'
       FROM Inserted i
       JOIN Deleted d
          ON i.ProductID = d.ProductID
       WHERE (d.UnitsInStock - i.UnitsInStock) > d.UnitsInStock / 2
          AND d.UnitsInStock - i.UnitsInStock > 0
      )
   BEGIN
      RAISERROR('Cannot reduce stock by more than 50%% at once.',16,1)
      ROLLBACK TRAN
   END
```

Прежде чем приступить к проверке работы этого триггера, проанализируем, какие действия в нем выполняются.

На первом этапе проверяется условие `IF EXISTS`, как и во многих других примерах настоящей главы. Единственная цель этой проверки состоит в том, чтобы выполнить откат, если поступивший запрос соответствует этим ограничительным, лишающим свободы выбора и неприятным критериям, которые мы обязаны соблюдать.

Затем создается соединение таблиц `INSERTED` и `DELETED`, что позволяет сравнить данные этих двух таблиц.

Определенную сложность может представлять конструкция `WHERE`, в первой строке которой содержится довольно простое выражение. В данном выражении воплощается словесная формулировка реализуемого делового требования, согласно которому обновления в столбце `UnitsInStock`, превышающие по объему половину количества единиц товара, которые прежде имелись в наличии, соответствуют заданному критерию, поэтому при их обнаружении транзакция должна быть отвергнута.

Но следующая строка не столь проста. Как и при решении многих других задач программирования, мы обязаны выйти за рамки словесной формулировки задачи и понять, к чему приводит реализация этой формулировки в программе. Фактически предъявленное требование касается только случаев сокращения объема товарных запасов в результате выполнения заказов на товары. Но, безусловно, никто не стремится ввести ограничение на то, какое количество единиц товара можно будет поместить на склад, поэтому необходимо предусмотреть, чтобы ограничение коснулось только таких обновлений, в результате которых количество товара после обновления становится меньше, чем до обновления.

Если соблюдаются оба условия (изменение объема запасов больше чем на 50%, а также сокращение, а не увеличение запасов), то активизируется ошибка. Обратите внимание на то, что в вызове функции RAISERROR заданы два знака %, а не один. Напомним, что знак % используется как метка-заполнитель для параметра, поэтому если будет задан только один знак %, то после вывода сообщения об ошибке он будет отсутствовать в тексте. Помещая в строку подряд два знака процента, %%, мы указываем СУБД SQL Server, что в данном случае требуется вывести на печать один знак процента.

Теперь проверим работу триггера. Предположим, что сотрудник торгового отдела только что ввел следующий оператор и пытается с его помощью сократить объем товарных запасов больше чем на 50%:

```
UPDATE Products
   SET UnitsInStock = 2
   WHERE ProductID = 8
```

В данном случае экспериментам подвергается товар "Northwoods Cranberry Sauce", но можно было бы взять для проверки товар с любым идентификатором ProductID, при условии, что заданное значение будет составлять меньше 50% предыдущего значения. При попытке ввода этого оператора формируется сообщение об ошибке, как и следовало ожидать:

```
Msg 50000, Level 16, State 1, Line -1074284106
Cannot reduce stock by more than 50% at once.
```

Следует отметить, что можно было бы также попытаться решить указанную задачу с помощью таблицы Order Details, сравнивая фактический объем заказа с текущим значением количества запасов UnitInStock, но при этом пришлось бы столкнуться с некоторыми описанными ниже проблемами.

- Изменяющиеся обновления. Неизвестно, происходит ли в процессе создания строки таблицы Order Details обновление данных таблицы Products до или после вставки строки в таблицу Order Details. От этого зависит решение, как должно использоваться значение UnitsInStock в таблице Products для вычисления результатов транзакции.
- Отсутствие контроля над изменениями товарных запасов, не связанными с вводом данных в таблицу Order Details. Сокращение товарных запасов больше чем наполовину может происходить в результате выполнения каких-либо других операций (во многих приложениях подобная возможность вполне допустима, но соблюдение ограничений необходимо обеспечить независимо от принятой организации работы).

Использование триггеров для формирования определяемых пользователем сообщений об ошибках

Тема формирования определяемых пользователем сообщений об ошибках уже рассматривалась в настоящей книге на нескольких других примерах, но следует отметить, что триггеры могут оказаться очень удобными в тех случаях, когда необходимо обеспечить контроль над формируемыми сообщениями об ошибках или кодами ошибок, которые передаются пользователю или поступают в клиентское приложение.

В частности, если используются ограничения CHECK, то приходится довольствоваться тем, что при обнаружении недопустимой ситуации активизируется стандартная ошибка с кодом 547, которая сопровождается весьма маловыразительным описанием. Чаще всего применение такого способа активизации ошибок практически не может сообщить пользователю, чтобы он мог понять, почему все происходит не так, как надо. В действительности в клиентском приложении часто отсутствует достаточный объем информации для того, чтобы можно было сформировать обоснованный и разумный отклик от имени пользователя.

Короче говоря, иногда создаются такие триггеры, которые на деле позволяют решить поставленные задачи обеспечения целостности данных, но не предоставляют достаточного объема информации для того, чтобы можно было устранить причины возникшего нарушения в работе.

Другие распространенные области применения триггеров

Триггеры используются не только непосредственно для обеспечения целостности данных, но и имеют целый ряд других областей применения. В действительности открывающиеся возможности использования триггеров практически неограничены, но ниже приведены некоторые наиболее часто встречающиеся примеры применения триггеров.

- Обновление итоговой информации.
- Подготовка денормализованных таблиц, предназначенных для формирования отчетов.
- Выставление флажков для проверки условий.

Вполне очевидно, что возможности применения триггеров являются весьма разнообразными. В действительности выбор способа использования триггеров зависит исключительно от конкретной ситуации и потребностей определенной прикладной системы.

Другие вопросы, связанные с использованием триггеров

В предыдущих разделах были приведены основные сведения о триггерах, позволяющие непосредственно приступить к их использованию, а в этом разделе описано применение триггеров. Как было указано в начале главы, многие вопросы, связанные с применением триггеров, требуют тщательного анализа. В этом разделе предпринята попытка рассмотреть наиболее важные проблемы, которые необходимо учитывать при использовании триггеров, а также предоставить некоторую информацию о дополнительных средствах и возможностях триггеров.

Применение вложенных триггеров

Вложенным триггером называется такой триггер, запуск которого происходит не в качестве непосредственного результата вызова на выполнение некоторого оператора, а в результате выполнения оператора, вызванного в другом триггере.

В связи с применением вложенных триггеров фактически могут возникать целые цепочки событий, если запуск одного триггера приводит к запуску другого триггера, который, в свою очередь, вызывает запуск еще одного триггера, и т.д. Возможная степень вложенности, возникающая при запуске триггеров, зависит от описанных ниже факторов.

- От того, разрешено ли использование вложенных триггеров в системе (эта опция определяется на уровне системы, а не базы данных; для ее определения применяется программа Management Studio или процедура `sp_configure`, и по умолчанию она разрешена).
- От установленного лимита вложенности, составляющего 32 уровня.
- От того, произошел ли уже запуск триггера. По умолчанию запуск триггера может происходить только один раз в расчете на каждую транзакцию с триггером. После запуска триггера игнорируются все прочие вызовы, возникающие в результате осуществления операций, составляющих часть одного и того действия триггера. После перехода к обработке полностью нового оператора в процессе (даже в пределах одной и той же общей транзакции) вся обработка может начаться сначала.

В большинстве обстоятельств действительно возникает необходимость в использовании вложенных триггеров (именно поэтому такая организация работы применяется по умолчанию), но необходимо учитывать, что может произойти, если возникнет заколдованный круг, в котором запуск одного триггера за другим становится нескончаемым. С другой стороны, если снова произойдет повторный возврат к одной и той же таблице, то во второй раз запуск триггера не будет осуществлен, и может оказаться, что не выполнены какие-либо важные действия, а из-за этого, например, может возникнуть нарушение целостности данных. Следует также отметить, что при обнаружении оператора `ROLLBACK` в любом месте цепочки вложения происходит откат всей цепочки. Иными словами, вся цепочка вложенных триггеров действует как одна транзакция.

Рекурсивный вызов триггеров

Триггеры допускают рекурсивный вызов, если содержат такой код, выполнение которого в конечном итоге может вызвать запуск того же триггера. Рекурсивный запуск может происходить непосредственно (в результате действия запроса, применяемого в таблице, на которой задан триггер) или опосредованно (в результате осуществления процесса запуска вложенных триггеров).

Рекурсивные вызовы триггеров применяются редко. И действительно по умолчанию возможность рекурсивного вызова триггеров не предусматривается. Тем не менее рекурсивный запуск триггеров позволяет найти выход из только что описанной ситуации, в которой применяется вложение триггеров и требуется, чтобы обновление таблицы было выполнено дважды. В отличие от опции, допускающей использование вложения, опция, определяющая возможность применения рекурсии, уста-

навливается на уровне базы данных; для этого может служить хранимая системная процедура sp_dboption.

Применение рекурсивного вызова триггеров связано с такой опасностью, что в той или иной форме может быть непреднамеренно образован цикл. Из этого следует, что в приложении должен быть предусмотрен определенный способ проверки рекурсии, позволяющий в случае необходимости остановить процесс рекурсивного вызова.

Отсутствие возможности предотвратить с помощью триггеров внесение структурных изменений

Триггеры могут применяться для внесения изменений в структуру базы данных, и в этой области наиболее ярко проявляются их преимущества и недостатки.

Преимущества триггеров состоят в том, что их использование не препятствует внесению изменений в структуру базы данных. И действительно разработчики часто используют триггеры для обеспечения ссылочной целостности на ранних этапах цикла разработки (когда еще очень велика вероятность того, что придется вносить большое количество изменений в проект базы данных), а затем переходят к применению декларативных ограничений ссылочной целостности в тот период жизненного цикла приложения, который находится гораздо ближе к этапу передачи на производство.

Если используются декларативные средства обеспечения ссылочной целостности, то для уничтожения и повторного создания таблицы необходимо вначале уничтожить все ограничения и только после этого удалить саму таблицу. В связи с этим может потребоваться выполнить огромный объем работы, необходимый для удаления многочисленных ограничений, внесения изменений, а затем повторного ввода в действие всех ограничений. А если эти операции должны быть выполнены достаточно быстро, то может возникнуть чрезвычайно напряженная рабочая ситуация, поскольку приходится контролировать, чтобы были действительно своевременно уничтожены все намеченные для этого объекты и модифицированные сценарии выполнены успешно. А затем возникает не менее напряженная ситуация, поскольку необходимо обеспечить своевременное восстановление всех требуемых объектов. При использовании триггеров все эти задачи решаются гораздо проще, поскольку любые структурные изменения могут проводиться беспрепятственно, ведь они обнаруживаются только после фактического запуска триггера.

Но из сказанного выше следует одно предостережение, поскольку в триггерах структурные изменения обнаруживаются только после их запуска, а это означает, что некоторые структурные изменения могут приводить к нарушению работы многочисленных триггеров, но это даже не будет обнаружено. Вернее, такое нарушение в работе останется незамеченным до тех пор, пока в тех же триггерах не будет предпринята попытка обратиться к модифицированному объекту (объектам), после чего и появится ошибка, вызванная не до конца продуманными изменениями. Вот тогда-то и возникнут трудности при попытке полностью уяснить для себя, что же было сделано и почему.

Таким образом, при внесении изменений в структуру базы данных сложности возникают и тогда, когда используются декларативные ограничения ссылочной целостности и когда используются триггеры, поэтому необходимо учитывать специфику применения тех и других методов поддержки ссылочной целостности.

Отмена действия триггеров без их удаления

Иногда, как и при использовании ограничений CHECK, возникает необходимость отменить на время действие средств поддержки целостности данных, чтобы можно было выполнить какие-либо операции, нарушающие заданные ограничения, но без которых невозможно обойтись (по-видимому, наиболее распространенным примером таких операций является импорт данных).

Еще одной часто возникающей причиной отмены ограничений является осуществление массовой вставки данных того или иного типа (это — также разновидность импорта), но уже при полной уверенности в том, что все данные являются допустимыми. В этом случае может потребоваться отменить действие триггеров, чтобы устранить издержки, связанные с их запуском, и ускорить процесс вставки.

Для ввода в действие и отмены действия триггеров можно применить оператор ALTER TABLE, который имеет следующий синтаксис:

```
ALTER TABLE <table name>
  <ENABLE|DISABLE> TRIGGER <ALL|<trigger name>>
```

Вполне очевидно, что наиболее важное предостережение при использовании такого метода состоит в том, что не следует забывать снова вводить триггеры в действие.

Еще одно важное соображение состоит в следующем. Если происходит отмена действия триггеров для осуществления в той или иной форме массового импорта данных, то следует настоятельно порекомендовать, чтобы были закрыты сеансы всех пользователей и выполнен переход в однопользовательский режим, режим доступа только для владельца базы данных (dbo) или в режим, характеризующийся обоими указанными признаками. Это позволяет исключить возможность ввода в базу данных недопустимой информации посторонними людьми в тот период, когда триггеры не действуют.

Порядок запуска триггеров

SQL Server 7.0 и предыдущие версии не позволяли управлять порядком запуска триггеров. И действительно, как уже было сказано выше, до появления версии 7.0 допускалось применение одновременно только одного триггера определенного типа (INSERT, UPDATE или DELETE), поэтому не было смысла вести речь о порядке запуска триггеров. Но в последующих версиях SQL Server был предусмотрен ограниченный объем средств контроля над тем, какие триггеры должны запускаться и в какой последовательности. Для каждой конкретной таблицы (но не представления, поскольку порядок запуска может быть задан только для триггеров AFTER, а представления допускают использование лишь триггеров INSTEAD OF) разрешалось выбрать один (и только один) триггер, запускаемый в первую очередь. Аналогичным образом была предусмотрена возможность выбрать один (и только один) триггер, запускаемый в последнюю очередь. Все остальные триггеры рассматривались как не имеющие приоритета по отношению к порядку запуска. Иными словами, нельзя было гарантировать, в каком порядке будет запускаться триггер, для которого задан порядок запуска "NONE" (не определено). Известно было лишь, что такие триггеры (рис. 15.3) должны быть запущены после завершения первого триггера, FIRST (если он имеется), и до начала работы последнего триггера, LAST (также если он имеется).

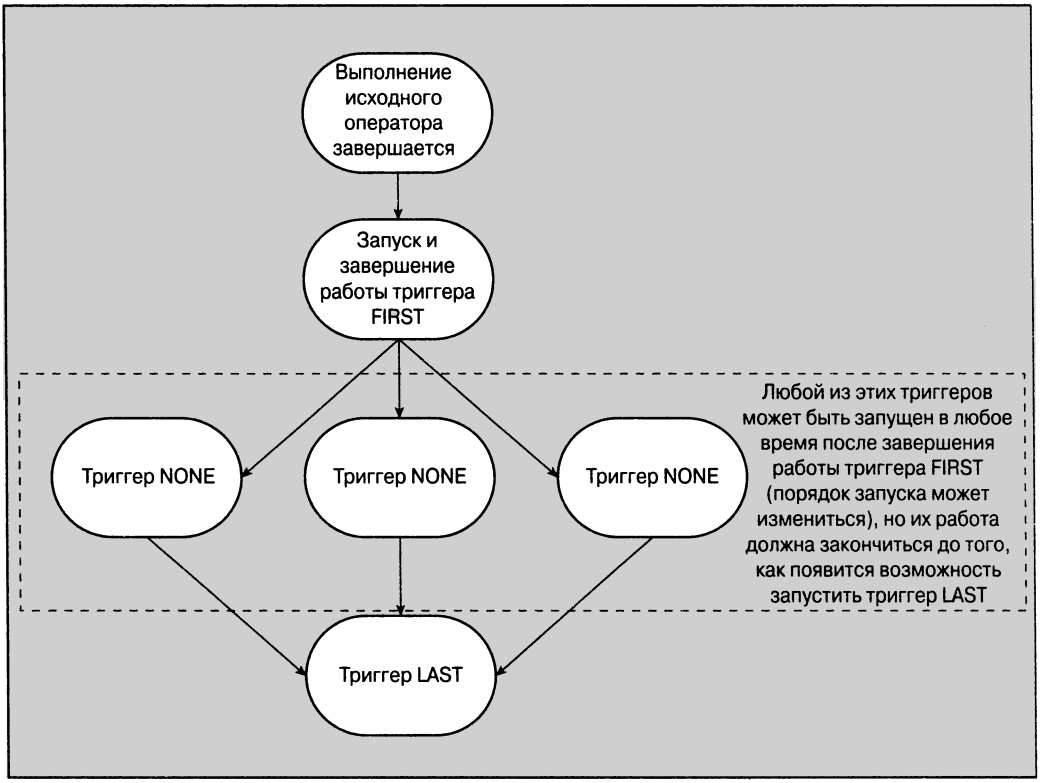


Рис. 15.3. Порядок запуска триггеров

При создании триггера, который должен запускаться в первую или последнюю очередь, не предъявляются какие-либо особые требования по сравнению с любыми другими триггерами. Единственное различие между триггерами с точки зрения порядка их запуска состоит в том, что для них задаются разные значения приоритета запуска после их создания с использованием специальной системной хранимой процедуры `sp_settriggerorder`.

Вызов хранимой процедуры `sp_settriggerorder` имеет следующий синтаксис:

```
sp_settriggerorder[@triggername =] '<trigger name>',
   [@order =] '{FIRST|LAST|NONE}',
   [@stmttype =] '{INSERT|UPDATE|DELETE}'
```

Для каждого конкретного действия (`INSERT`, `UPDATE` или `DELETE`) должен быть задан только один триггер, который рассматривается как имеющий обозначение "`FIRST`". Аналогичным образом для каждого конкретного действия задается только один триггер с обозначением "`LAST`". А обозначение "`NONE`" может быть присвоено любому количеству триггеров. Иными словами, количество триггеров, для которых не задан определенный порядок запуска, является неограниченным.

Из сказанного следует, что необходимо также рассмотреть вопрос о том, имеет ли значение порядок запуска других триггеров. Достаточно часто складываются ситуации, в которых порядок запуска остальных триггеров не имеет особого значения.

Но иногда от того, в каком порядке происходит запуск триггеров, зависит успешная реализация алгоритмов приложения или обеспечение приемлемой производительности. Рассмотрим эти вопросы подробнее.

Управление порядком запуска для обеспечения правильной реализации алгоритмов

Необходимость обеспечить запуск одного триггера перед другим чаще всего может быть обусловлена тем, что с помощью ранее запущенного триггера закладывается своего рода фундамент или, вообще говоря, создаются предпосылки осуществления дальнейших действий. При эксплуатации SQL Server 6.5 и предыдущих версий в основном не приходилось задумываться о подобных проблемах, поскольку для каждой конкретной таблицы разрешалось задавать только по одному триггеру каждого конкретного типа (`UPDATE`, `DELETE` или `INSERT`). Из этого следовало, что фактически не составляло труда обеспечить запуск триггера одного типа прежде другого. А поскольку в триггере были реализованы все связанные с его использованием алгоритмы обработки, достаточно было предусмотреть в коде триггера выполнение в первую очередь тех операций, которые должны стоять на первом месте, а операции, выполняемые в последнюю очередь, поместить на последнее место (таким образом, значительные сложности не возникали).

После выхода версии 7.0 положение дел одновременно и улучшилось, и ухудшилось. С одной стороны, отпала необходимость помещать весь код реализации всех алгоритмов в один триггер. Это оказалось удобным, поскольку позволило физически разделить части кода триггера, относящиеся к реализации разных процедур обработки данных, что позволило значительно упростить управление кодом и выборочно отменять на время действие отдельных частей кода (как было сделано в одном из предыдущих разделов с помощью опции `NO CHECK`), тогда как другие части кода продолжали функционировать. С другой стороны, обнаруживался недостаток, обусловленный тем, что после разделения кода таким образом исключалась возможность сохранить логическую последовательность реализации в коде алгоритмов поэтапной обработки, которые когда-то можно было реализовать в единственном триггере.

Но если есть возможность обеспечить контроль над последовательностью запуска хотя бы на элементарном уровне, то приложение приобретает все возможные преимущества старых и новых версий, поскольку обеспечивается реализация алгоритмов с помощью отдельных триггеров и вместе с тем сохраняется возможность задать необходимый порядок предшествования, позволяющий указать, какие фрагменты кода должны выполняться в первую или последнюю очередь.

Управление порядком запуска для обеспечения производительности

Триггером единственного типа, с помощью которого удается добиться существенного повышения производительности, является триггер `FIRST`. Для этого при наличии нескольких триггеров, таких что лишь один из них, по всей вероятности, может вызвать откат (например, из-за того, что вводится в действие сложное правило обеспечения целостности данных, которое невозможно реализовать с помощью ограничения), именно этот триггер должен быть предназначен для запуска в первую очередь как триггер `FIRST`. Это позволяет добиться того, чтобы наиболее вероятная причина отката уже осталась позади до того, как будут приложены какие-то дополнительные усилия в осуществление транзакции. Ведь чем больший объем работы будет

выполнен до обнаружения условий выполнения отката, тем больше придется сделать в процессе выполнения отката. Поэтому, прежде чем переходить к осуществлению дальнейших действий, необходимо сделать все возможное для того, чтобы был своевременно обнаружен возможный откат.

Триггеры INSTEAD OF

Триггеры INSTEAD OF были впервые введены в версии SQL Server 2000 и с тех пор остаются одним из наиболее сложных средств SQL Server. Безусловно, описание триггеров этого типа далеко выходит за рамки тех понятий, с которыми необходимо ознакомить начинающих разработчиков, но автор твердо уверен в том, что даже начинающие должны узнать о том, какие возможности находятся в их распоряжении, поэтому в данном разделе представлены некоторые вводные сведения о триггерах этого типа.

Триггер INSTEAD OF представляет собой блок кода, который может использоваться для выполнения определенных операций в качестве замены любых других операций, которые кто-то другой может попытаться применить к таблице или представлению. Триггеры этого типа позволяют сделать выбор: продолжить ли работу и выполнить все, что требует пользователь, или, в случае необходимости, осуществить совсем другие действия.

Триггеры INSTEAD OF, как и обычные триггеры, подразделяются на три разновидности: INSERT, UPDATE и DELETE. В любом случае основная область применения этих триггеров остается одной и той же — устранение неопределенности в отношении того, какая таблица (таблицы) должна получать обновления, если работа с базой данных ведется с помощью представления, основанного на нескольких таблицах.

Рекомендации по повышению производительности триггеров

Специалисты по базам данных еще не достигли единого мнения по поводу преимуществ и недостатков триггеров. Наиболее ожесточенные споры происходят между теоретиками, которые, обосновывая необходимость отказа от триггеров, опираются в основном на умозрительные соображения, и практиками, которым действительно удалось решить с помощью триггеров очень сложные задачи, после чего они, по-видимому, стремятся использовать триггеры чуть ли не повсеместно.

Мнение автора по этому вопросу, которое уже было высказано в настоящей главе, состоит в том, что триггеры следует использовать строго по назначению. Я не буду чувствовать неловкость, если кто-то отметит, что мое мнение выражено уклончиво и неоднозначно. В программировании и тем более в области разработки приложений для баз данных часто невозможно указать единственно правильный подход. Тем не менее в следующих разделах приведены некоторые факты для размышлений.

Выполнение триггеров с отставанием, а не с опережением

Триггеры отличаются от таких средств обеспечения целостности данных, как ограничения, тем, что выполняются с отставанием, а не с опережением. Под этим подразумевается то, что запуск триггера происходит после осуществления тех действий,

с которыми связан триггер. Ко времени запуска триггера завершаются выполнение всего запроса и регистрация транзакции в журнале (но фиксация не происходит и регистрируется только часть транзакции, предшествующая оператору, который вызвал запуск триггера). Это означает, что если возникнет необходимость осуществить откат в связи с выполнением кода триггера, то может оказаться, что из-за этого придется отменить результаты огромного объема работы, которая уже была проделана, а это приводит к значительному замедлению. Но из-за этого не следует полностью отказываться от использования триггеров, ведь степень влияния таких последствий существенно зависит от того, насколько большой объем работы предусмотрен в запросе.

Рассмотрим, какие следствия вытекают из сказанного. Прежде всего любопытно сравнить то, как используются триггеры, с применением ограничений, которые применяются заблаговременно; иными словами, проверка ограничений происходит до того, как действительно начинается выполнение оператора. Это означает, что при использовании ограничений в процессе обработки данных заблаговременно обнаруживаются факторы, которые должны стать причиной неудачного завершения, поэтому такое развитие ситуации заранее предотвращается. В связи с этим программное обеспечение, в котором применяются ограничения, обычно функционирует пусть даже ненамного, но быстрее, чем при использовании триггеров, причем по мере усложнения запросов различие в быстродействии увеличивается. Но следует отметить, что такое повышение быстродействия фактически становится достаточно заметным только в тех ситуациях, когда возникают предпосылки применения отката.

На основании сказанного выше можно сделать следующий вывод. Если в процессе работы приложения приходится сталкиваться с очень небольшим относительным количеством откатов, а сложность и продолжительность прогона операторов, подверженных откату, невелики, то, по-видимому, различия в быстродействии между триггерами и ограничениями не столь существенны. Разумеется, такие различия всегда обнаруживаются, но, по всей вероятности, не оказывают слишком большого влияния. Но если количество возможных откатов заранее не предсказуемо, а тем более, если известно, что оно должно быть велико, то по возможности следует придерживаться методов обеспечения целостности, основанных на использовании ограничений (и, откровенно говоря, автор рекомендовал бы всегда строить свою работу по обеспечению целостности данных на основе ограничений, если нет достаточно весомых оснований для принятия другого решения).

Отсутствие проблем при организации параллельной работы триггеров и процессов, в которых они активизируются

По-видимому, читатель обнаружил во многих примерах, приведенных в данной главе, что в них часто используется оператор `ROLLBACK`, даже несмотря на то, что соответствующий оператор `BEGIN TRAN` на выполнение не вызывается. Это связано с тем, что триггер всегда неявно становится частью той же транзакции, что и оператор, который вызвал запуск триггера.

Если оператор, в котором запускается триггер, не является частью явно заданной транзакции (т.е. транзакции, в которой имеется оператор `BEGIN TRAN`), то подобный оператор все равно входит в состав собственной транзакции, состоящей из одного оператора. Но в любом случае вызов оператора `ROLLBACK TRAN` внутри триггера приводит к откату всей транзакции.

Еще одним следствием из того, что триггеры рассматриваются как часть одной и той же транзакции, становится наследование триггерами всех блокировок, уже открытых в транзакции, в состав которой они входят. Это означает, что в коде триггера не требуется предпринимать каких-либо особых действий для предотвращения возможности конфликта с блокировками, созданными в других операторах той же транзакции. В коде триггера предоставляется свободный доступ ко всем объектам данных, принадлежащим к области определения транзакции, кроме того, в коде триггера наблюдается такое состояние базы данных, в котором учтены все модификации, внесенные в данные предыдущими операторами транзакции.

Использование функций UPDATE() и COLUMNS_UPDATED()

Триггер UPDATE часто позволяет ограничить объем кода, фактически выполняемого в триггере, путем проверки, осуществляемой для определения того, является ли интересующий нас столбец (столбцы) одним из тех столбцов, которые подвержены изменению. Для этой цели предназначены функции UPDATE() и COLUMNS_UPDATED(). Описания этих функций приведены в следующих разделах.

Функция UPDATE()

Применение функции UPDATE() имеет смысл только в области определения триггера. Единственное назначение этой функции состоит в том, что с ее помощью выясняется, обновлялся ли какой-то конкретный столбец или нет. Функция возвращает результат проверки в виде булева значения (true или false). Эта функция позволяет определить, должен ли быть вызван на выполнение какой-то конкретный блок кода, например, код, применение которого имеет смысл, только если обновляется какой-то определенный столбец.

Рассмотрим краткий пример применения функции UPDATE(), представляющий собой один из описанных ранее триггеров, в который внесены соответствующие изменения:

```
ALTER TRIGGER ProductIsRationed
   ON Products
   FOR UPDATE
AS
   IF UPDATE(UnitsInStock)
   BEGIN
   IF EXISTS
      (
       SELECT 'True'
       FROM Inserted i
       JOIN Deleted d
          ON i.ProductID = d.ProductID
       WHERE (d.UnitsInStock - i.UnitsInStock) > d.UnitsInStock / 2
          AND d.UnitsInStock - i.UnitsInStock > 0
      )
   BEGIN
      RAISERROR('Cannot reduce stock by more than 50%% at once.',16,1)
      ROLLBACK TRAN
   END
   END
```

Доработка этого кода позволяет регламентировать выполнение остальной части кода с учетом того, касаются ли изменения в данных только столбца `UnitsInStock` (который представляет для нас интерес). Таким образом, пользователю предоставляется возможность вносить изменения в любые другие столбцы, но в связи с этим не выполняется какой-либо код. Это означает, что благодаря применению функции `UPDATE()` количество выполняемых строк кода триггера сокращается, поэтому быстродействие триггера по сравнению с предыдущей версией немного повышается.

Функция COLUMNS_UPDATED()

Функция `COLUMNS_UPDATED()` действует немного иначе, чем функция `UPDATE()`, но имеет такое же общее назначение. Дополнительные возможности функции `COLUMNS_UPDATED()` обусловлены тем, что эта функция позволяет проверить, обновляются ли одновременно несколько столбцов. Для этого в функции применяется битовая маска, которая связывает отдельные биты в одном или нескольких байтах данных типа `varbinary` с отдельными столбцами таблицы. В результате вызова функции `COLUMNS_UPDATED()` формируется битовая маска, которая выглядит примерно так, как показано на рис. 15.4.

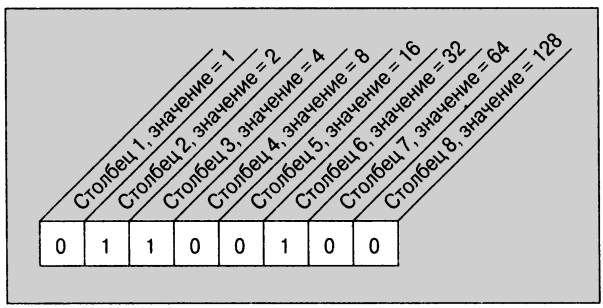


Рис. 15.4. Битовая маска, которая включает биты одного байта

На рис. 15.4 видно, что битовая маска состоит из одного байта данных, который содержит информацию о том, что обновлены данные во втором, в третьем и шестом столбцах, а остальные столбцы остались незатронутыми.

В том случае, если количество столбцов превышает восемь, шестнадцать и т.д., в СУБД SQL Server добавляется еще один байт справа и отсчет столбцов продолжается (рис. 15.5).

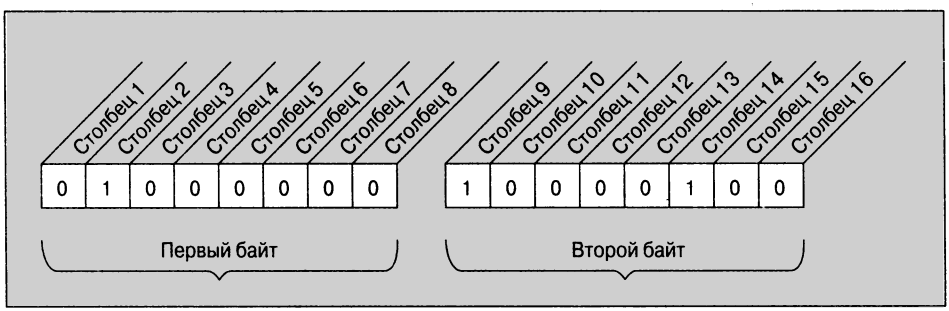


Рис. 15.5. Битовая маска, которая включает биты двух байтов

Битовая маска (см. рис. 15.5) свидетельствует о том, что обновлены второй, девятый и четырнадцатый столбцы.

Для обработки результатов, полученных с помощью функции `COLUMNS_UPDATED()`, применяются логические операторы.

Для этого прежде всего необходимо подготовить контрольную битовую маску (представленную в виде десятичного числа), сложив двоичные значения всех битов, отсчитываемые слева направо. Таким образом, если необходимо определить, было ли выполнено обновление столбцов 2, 5 и 7, необходимо сложить двоичные значения битов, соответствующих этим столбцам: `2 + 16 + 64`. После этого результаты вызова функции `COLUMNS_UPDATED()` сравниваются с полученной контрольной битовой маской с помощью описанных ниже операторов побитовой обработки.

- `|`. Побитовая операция "ИЛИ".
- `&`. Побитовая операция "И".
- `^`. Побитовая операция "исключительное ИЛИ".

Рассмотрим несколько примеров.

Предположим, что выполнено обновление таблицы, которая содержит пять столбцов. Если необходимо узнать, обновлены ли первый, третий и пятый столбцы, то следует прежде всего подготовить битовую маску, предназначенную для проверки возвращаемого значения функции `COLUMNS_UPDATED()`, которая содержала бы двоичное значение 10101000, или десятичное значение 1 + 4 + 16 = 21. После этого можно применить приведенные ниже выражения.

- `COLUMNS_UPDATED() > 0`. Это выражение позволяет узнать, был ли вообще обновлен какой-либо столбец.
- `COLUMNS_UPDATED() ^ 21 = 0`. С помощью этого выражения можно определить, произошло ли обновление только указанных столбцов (в этом случае столбцов 1, 3 и 5).
- `COLUMNS_UPDATED() & 21 = 21`. Это выражение позволяет выяснить, произошло ли обновление всех указанных столбцов, если состояние всех прочих столбцов не представляет интереса.
- `COLUMNS_UPDATED | 21 != 21`. Выражение, которое дает возможность узнать, был ли обновлен какой-либо столбец, кроме тех, которые нас интересуют.

В выражения с логическими операторами может слишком легко закрасться ошибка, поэтому такие выражения должны подвергаться тщательной проверке и тестированию.

Применение триггеров с небольшим объемом кода

Безусловно, рекомендации, приведенные в этом разделе, представляют собой повторение ранее высказанных рекомендаций, но автор обязан их повторить, поскольку для этого есть весомые основания.

Сложно даже представить себе, насколько часто в процессе изучения существующих программ приходится сталкиваться с хранимыми процедурами и триггерами, содержащими раздутый и непродуманный код. Нелегко понять, чем руководствуются программисты, создающие подобные программы, — либо они очень спешат, либо считают, что СУБД и так работает достаточно быстро, поэтому не нужно заботиться о повышении быстродействия.

Следует помнить, что триггер — это такая же часть транзакции, как и оператор, из которого он вызван. Это означает, что выполнение оператора транзакции не заканчивается до тех пор, пока не завершится выполнение триггера. Из этого следует, что если для выполнения кода, содержащегося в триггере, потребуется много времени, то, в свою очередь, будет занимать много времени выполнение каждого фрагмента созданного вами кода, который вызывает запуск триггера. В связи с этим возникают буквально непреодолимые трудности при попытке понять, почему на выполнение кода затрачивается так много времени. На первый взгляд кажется, что написанная хранимая процедура должна быть чрезвычайно эффективной, но на самом деле обнаруживается, что ее производительность буквально неприемлема. Вы можете биться над этой проблемой неделями, но так и не понять, что проблема заключается не в самой хранимой процедуре, — в ней просто происходит запуск триггера, имеющего чрезвычайно низкую производительность.

Выбор индексов с учетом наличия триггеров

Еще одна распространенная ошибка заключается в том, что при выборе индексов забывают о существовании триггеров. Разработчик просматривает все свои хранимые процедуры и представления, определяя, какое сочетание индексов является наилучшим, и полностью забывает о том, что значительная часть кода выполняется в виде триггеров.

В данном случае действует такой же принцип, который был описан в разделе "Применение триггеров с небольшим объемом кода", — в операторах вызываются запросы, выполнение которых занимает много времени, поэтому сами эти операторы долго не заканчивают свою работу, а это, в свою очередь, приводит к общему замедлению функционирования всех приложений. Занимаясь оптимизацией кода, не забывайте о триггерах!

Отказ от применения операторов отката в коде триггеров

Безусловно, эту рекомендацию нелегко осуществить на практике, поскольку откаты слишком часто составляют основную часть тех действий, которые требуются осуществить в триггерах.

Но следует помнить, что триггеры `AFTER` (которые бесспорно являются самым распространенным типом триггеров) вызываются на выполнение после того, как основная часть работы уже проделана, а это означает, что операция отката становится дорогостоящей. Именно в этих обстоятельствах декларативные средства поддержки целостности проявляют все свои преимущества, непревзойденные с точки зрения производительности. Если в триггерах приходится использовать много операторов `ROLLBACK TRAN`, то следует предусмотреть заблаговременную проверку на наличие возможных ошибок, прежде чем вызывать на выполнение оператор, в котором происходит запуск триггера. Иными словами, возьмите на себя заботу по осуществлению превентивных мер, поскольку сама СУБД SQL Server в этой ситуации их не осуществляет. Не дожидаясь отката, заранее проверяйте, будут ли возникать ошибки.

Удаление триггеров

Для удаления триггеров применяется такой же простой оператор, как и почти все описанные до сих пор операторы удаления:

```
DROP TRIGGER <trigger name>
```

После выполнения этого оператора триггер перестает функционировать.

Отладка кода триггеров

Попытка перейти в отладчике к триггеру по такому же принципу, как происходит переход к хранимой процедуре (см. главу 12) или функции, оканчивается неудачей, поскольку возможность непосредственной отладки кода триггера с помощью отладчика не предусмотрена. Тем не менее процедура отладки триггера является сложной, поэтому желательно применить для этой цели другой способ использования отладчика. Один из таких способов описан ниже. Безусловно, его нельзя назвать очень изящным, но он вполне применим.

Для реализации этого способа необходимо создать вспомогательную хранимую процедуру, в которой осуществляется запуск отлаживаемого триггера. Единственное назначение этой хранимой процедуры состоит в том, что она обеспечивает вызов на выполнение оператора, который позволяет войти в отладчике в код триггера.

В качестве примера рассмотрим фрагмент последней части проверочного кода, который использовался в данной главе, и поместим его в хранимую процедуру, чтобы можно было следить за тем, как происходит ее построчное выполнение в отладчике:

```
ALTER PROC spTestTriggerDebugging
AS
BEGIN
   -- Этот оператор должен быть выполнен успешно
   UPDATE Products
   SET UnitsInStock = UnitsInStock - 1
   WHERE ProductID = 6;

   -- А этот - нет
   UPDATE Products
   SET UnitsInStock = UnitsInStock - 12
   WHERE ProductID = 26;
END
```

После этого достаточно перейти к этой хранимой процедуре в окне Server Explorer, щелкнуть на ее обозначении правой кнопкой мыши и выбрать команду Step Into (рис. 15.6).

В открывшемся диалоговом окне щелкните на кнопке Execute (ввод каких-либо параметров не требуется). На первом этапе работы с отладчиком будут выполняться начальные операторы хранимой процедуры, но достаточно лишь выполнить команду Step Into для пошагового выполнения строк, которые вызывают запуск триггера, как будет получен приятный сюрприз (рис. 15.7), поскольку откроется код триггера!

Начиная с этого момента отладка с помощью отладчика может осуществляться обычным образом. Для этого применяются в основном такие же функции, которые были описаны в настоящей книге применительно к хранимым процедурам. Триггер даже становится активной частью стека вызовов.

Рекомендуем неизменно использовать описанный метод отладки, когда обнаруживаются столь распространенные ситуации, в которых так сложно происходит отладка кода триггеров. Безусловно, трудности встречаются и при использовании данного метода, но все же такая возможность лучше, чем ничего.

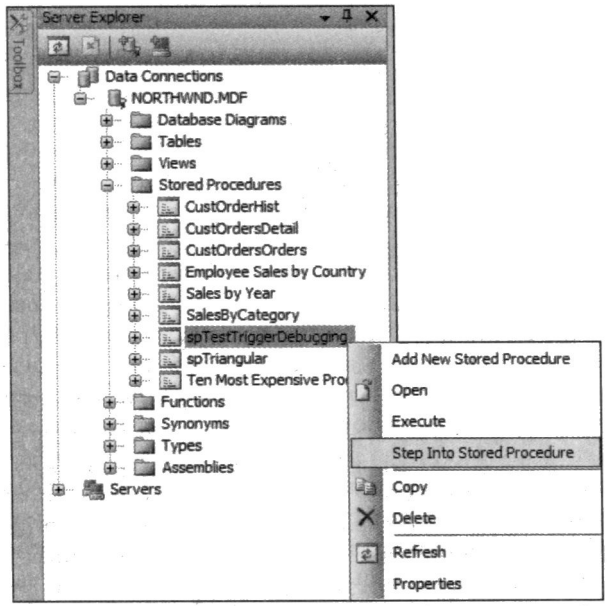


Рис. 15.6. Обозначение хранимой процедуры `spTestTriggerDebugging` *в окне* Server Explorer

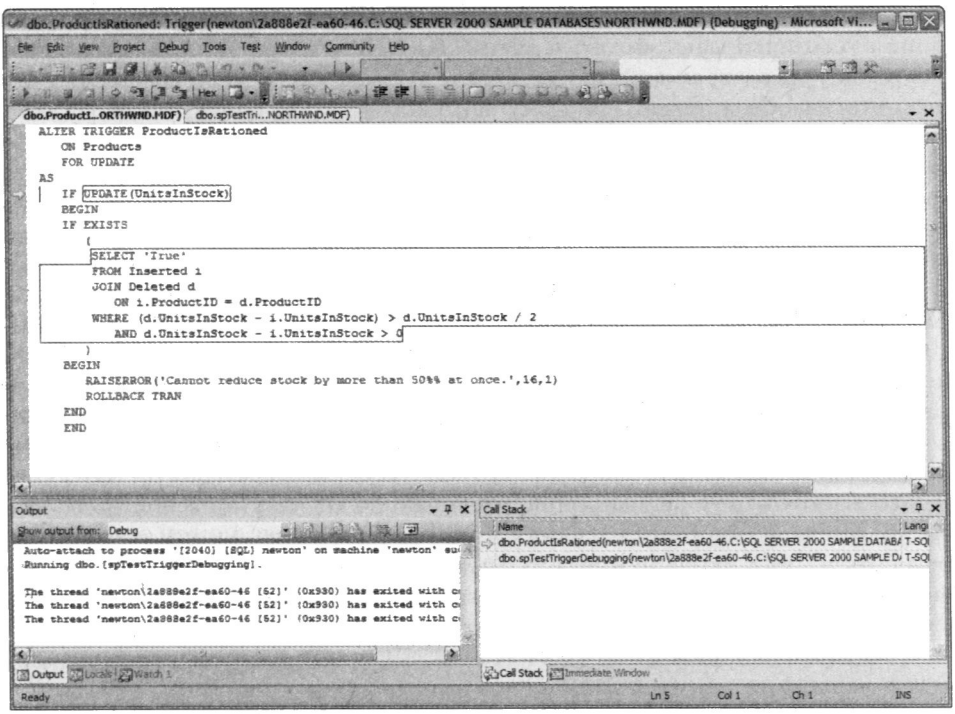


Рис. 15.7. Код триггера в окне отладчика

Резюме

Триггеры представляют собой чрезвычайно мощное инструментальное средство, поэтому позволяют существенно расширить функциональные возможности обеспечения целостности данных и в целом добиться более надежной эксплуатации всей системы. Но при использовании триггеров необходимо учитывать некоторые важные соображения. В частности, триггеры могут способствовать значительному повышению производительности системы, если они используются для обработки данных должным образом, но при неправильной организации работы вполне могут стать источником значительных затруднений. Отладка кода триггера может оказаться очень трудной задачей (даже после того, как появляется возможность осуществлять отладку с помощью отладчика), но решать эту задачу необходимо, поскольку из-за плохого качества кода триггера происходит снижение быстродействия при выполнении не только самого триггера, но и любого оператора, который вызывает запуск этого триггера.

Безусловно, задача разработки триггеров является весьма трудоемкой, даже если используются заранее подготовленные шаблоны триггеров, позволяющие упростить эту работу примерно на 90%. Но следует отметить, что многие инструментальные средства позволяют автоматически вырабатывать код триггеров, соответствующих определенным требованиям. Дополнительная информация по данной теме приведена в приложении В.

16

Краткий учебник по языку XML для начинающих

В предыдущих главах был изложен основной объем сведений, относящихся к наиболее важным компонентам СУБД SQL Server, а с этой главы начинается описание вспомогательных, но не менее важных компонентов. Иными словами, теперь мы приступаем к рассмотрению тех возможностей СУБД SQL Server, которые автор считает дополнительными. Сказанное не означает, что функциональные средства, которые предстоит рассмотреть в этой книге, обычно не применяются в той или иной отдельно взятой реляционной СУБД; речь идет лишь о том, что фактически эти средства не требуются для обеспечения нормальной работы СУБД SQL Server. А в действительности в настоящее время в состав программного обеспечения SQL Server включено столько компонентов, что их нелегко даже кратко описать в одной книге.

По-видимому, к категории наиболее важных дополнительных средств СУБД SQL Server относятся средства поддержки языка XML. В настоящей главе прежде всего приведены некоторые основные сведения по языку XML, после чего описаны многочисленные возможности, предусмотренные для поддержки языка XML. При этом автор стремится показать, что на самом деле язык XML представляет интерес сам по себе, а с его использованием связан подход, который полностью отличается от тех способов организации работы на основе реляционных систем, которые рассматривались выше в данной книге. Тем не менее в программном обеспечении SQL Server предусмотрено большое количество средств поддержки языка XML, и это связано с тем, что с внедрением языка XML произошло столь существенное расширение возможностей обработки данных, которое можно сравнить лишь с теми стремительными изменениями, которые были связаны с появлением хранилищ данных.

Со времени создания спецификации языка XML фактически прошло уже немало времени. Перспективы распространения XML с самого начала оценивались как

очень высокие, но в действительности распространение этого языка происходило не так быстро, как ожидалось. Однако начиная примерно с 2000 года язык XML стал все шире применяться как основа универсального способа обеспечения обмена данными и доступа к документам, содержащим значительный объем данных. Язык XML предоставляет возможность форматировать данные таким образом, чтобы они содержали свое описание в себе. Иначе говоря, предусмотрена возможность определять информацию о типах и структуре данных СУБД SQL Server, которая может передаваться вместе с документом XML, чтобы любой потребитель данных мог понять, по каким правилам сформированы данные (даже не зная ничего о том, как подключиться к самой СУБД).

Обычно документы XML не предоставляют наиболее экономичный способ хранения данных, но, безусловно, трудно найти другой подобный формат представления данных, который позволил бы обеспечить такое же удобство использования данных. Поэтому вполне можно рассчитывать на то, что сфера применения языка XML будет неограниченно расширяться.

Кроме всего сказанного, в настоящей главе будут рассматриваться следующие темы:

- общее описание языка XML;
- краткие сведения о технологиях, наиболее тесно связанных с языком XML.

Выше было сказано, что документы XML обычно не позволяют реализовать наиболее экономичный способ хранения данных, но из этого правила есть исключения. Как оказалось, язык XML может успешно использоваться для представления данных, предназначенных для хранения в архивах. Сжатие документов XML обеспечивается с очень высокой степенью эффективности, а сам язык XML позволяет создавать чрезвычайно удобные форматы, расшифровка которых не будет представлять никаких сложностей в течение многих предстоящих лет, возможно даже всех будущих веков. Эту возможность вполне можно оценить, получив, скажем, резервную копию в версии SQL Server 2005. Предположим, что со времени получения этой резервной копии пройдет десяток лет и потребуется восстановить некоторые старые данные для подготовки обзора архивной информации. Очевидно, что вряд ли удастся найти инсталляцию SQL Server, которая позволяла бы обрабатывать так давно полученные файлы с резервной копией, но весьма велика вероятность того, что сжатый архив можно будет распаковать (если для сжатия использовалась такая распространенная утилита, как ZIP) и прочитать архивированные данные. Таким образом, XML — весьма подходящий формат представления подобных "глубоких" архивов.

Основные сведения о языке XML

Описанию языка XML посвящено много хороших книг; в качестве примера можно назвать книгу Бирбека и др. (Birbeck et. al. *Professional XML*), выпущенную издательством Wrox. Эта книга содержит настолько полное описание XML, что вначале я решил отказаться от достаточно подробного описания самого языка XML, полагая, что читатель уже обладает определенными знаниями об этом языке. Но мне пришлось столкнуться с тем, что даже спустя много лет после выхода языка XML на магистральное направление развития чрезвычайно большое число специалистов по базам данных продолжают считать XML "просто какой-то Web-технологией" и поэтому не тратят время на его изучение; нелегко найти более неправильное мнение.

Язык XML – это основа информационных технологий, и его вообще не следует считать языком, относящимся исключительно к сфере Web-технологий. Тем не менее многие по-прежнему продолжают рассматривать XML под этим углом зрения (обычно это касается тех, кто не удосужился ознакомиться с XML) по нескольким причинам, описанным ниже.

- XML – это язык разметки, поэтому на первый взгляд чрезвычайно напоминает язык HTML.
- Документы XML чаще всего легко преобразуются в документы HTML. Поэтому язык XML как таковой послужил основой широко применяемого способа представления информационной части Web-страницы, в котором окончательное преобразование в формат HTML осуществляется только по запросу. При этом тот или иной формат преобразования может быть выбран с учетом определенных критериев (например, с учетом того, с какого браузера поступил запрос на получение страницы).
- Одним из первых широко применяемых программных продуктов, в котором была предусмотрена поддержка XML, оказался браузер Internet Explorer корпорации Microsoft.
- Интернет весьма часто используется как средство обмена информацией, а для этого идеально подходит язык XML.

Как и HTML, язык XML основан на использовании текстовой разметки, и это не случайно, поскольку оба этих языка были созданы на основе одного и того же оригинального языка, называемого SGML. Язык SGML был разработан задолго до того, как появился Интернет (по меньшей мере, та всемирная сеть, которую теперь мы называем Интернетом), и чаще всего использовался в издательском деле для представления печатной продукции или в правительственных учреждениях для оформления документации. Букву "S" в аббревиатуре SGML расшифровывают как simple (простой), но язык SGML является весьма трудно доступным для понимания и плохо поддается изучению. (Многие утверждают, что им никогда не удавалось прочесть больше 35% объема встречавшихся им документов SGML, но, с другой стороны, при чтении этих документов им неизменно удавалось добиться 100%-го отвращения.) В отличие от этого, документы на языке XML, как правило, являются весьма легко доступными для понимания.

Ознакомившись с этими предварительными сведениями, многие специалисты, владеющие языком HTML, начинают интересоваться тем, как можно получить список дескрипторов XML. Но такой список дескрипторов не существует; во всяком случае, он не является тем, с чем приходится сталкиваться при изучении других языков разметки. В состав самого языка XML фактически входит лишь очень немного дескрипторов. В действительности язык XML предоставляет лишь способы определения собственных дескрипторов, в том числе с использованием дескрипторов, которые определены другими (например, представителями промышленных групп, о которых шла речь выше в данной главе). В основе самого замысла создания XML лежало стремление обеспечить максимальное удобство пользования, в частности, предусмотреть возможность определять собственные правила форматирования документов XML либо с помощью определения схемы XML, либо с применением такого менее современного средства, как определение типа документа (Document Type Definition – DTD).

К самому документу XML предъявляется очень немного требований, обусловленных лишь тем, что он относится именно к этой категории. Наиболее важным из этих требований является обеспечение формальной правильности документа. Определение формально правильного документа будет приведено ниже, а пока что достаточно отметить, что документ XML должен в конечном счете не только соответствовать критериям формальной правильности, но и подходить под определение допустимого. Допустимым документом XML называется формально правильный документ XML, который соответствует также требованиям, предъявляемым к нему согласно схеме XML (или определению DTD), на которую ссылается документ. Сведения о схемах XML и определениях DTD приведены ниже в данной главе.

Еще одним важным свойством документов XML является то, что они легко поддаются преобразованию. Из этого следует, что документ XML может быть относительно просто преобразован с использованием полностью другого представления XML или даже формата, отличного от XML. Одним из наиболее широко применяемых направлений реализации этой возможности является преобразование кода XML в код HTML для отображения в Web-браузере. Дело в том, что язык HTML широко поддерживается как средство отображения в современных браузерах, но не позволяет столь же эффективно представлять информацию, как язык XML. Следует подчеркнуть, что сказанное выше является первым поводом для сравнения и сопоставления возможностей языков HTML и XML. Иными словами, язык XML в большей степени подходит для хранения информации, а язык HTML – для отображения информации.

Для хранения информации в документе XML используются структурные компоненты, называемые элементами и атрибутами. Элементы обычно создаются с применением открывающих и закрывающих дескрипторов (из этого правила есть исключение, но об этом будет сказано ниже) и обозначаются именами, чувствительными к регистру (использование пробелов в именах элементов не допускается). Атрибуты представляют собой синтаксические конструкции, позволяющие уточнять определения элементов, и располагаются в открывающем (или начальном) дескрипторе элемента. С атрибутами связаны значения, которые должны быть заключены в парные одинарные или двойные кавычки.

Части документа XML

В предыдущем разделе уже было приведено несколько терминов, но прежде чем перейти к более подробному описанию структуры документов XML, целесообразно провести общий обзор связанных с этим терминов и определений.

В связи с этим необходимо вначале описать все основные части документа XML (рис. 16.1). Многие части документа являются необязательными, но без некоторых частей невозможно обойтись. Кроме того, в одних случаях после включения в документ одной части требуется также включить другую, а другие структурные составляющие документа являются относительно независимыми друг от друга.

В целом документы XML имеют древовидную структуру, поэтому в настоящей главе для их описания принят в основном иерархический подход (структурные составляющие более низкого уровня, которые фактически являются вложенными в те части документа, которые относятся к более высокому уровню, рассматриваются в последнюю очередь), кроме того, если это имеет смысл, при описании частей документа учитывается порядок их следования в каком-то конкретном документе XML.

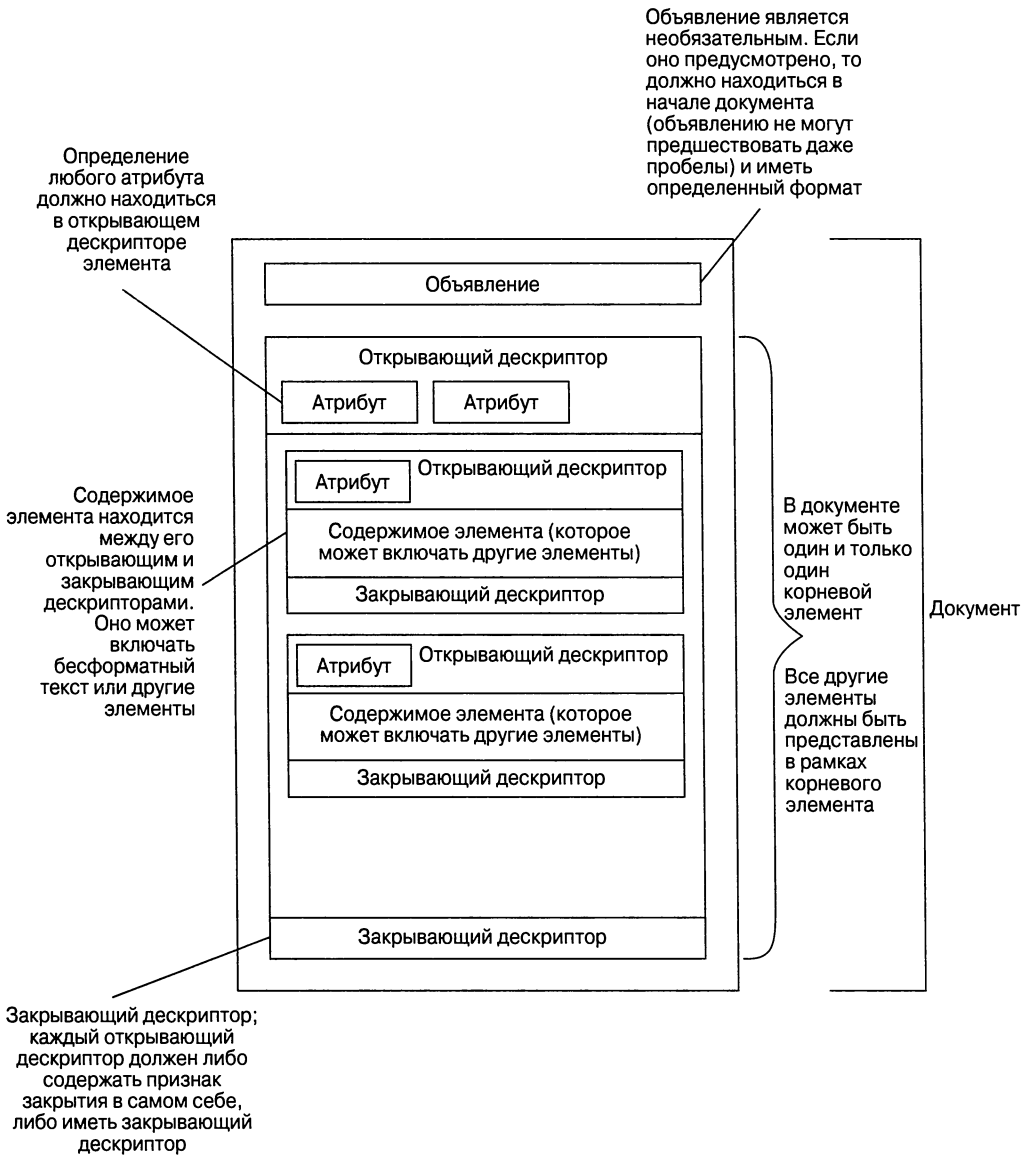


Рис. 16.1. Описание структуры документов XML

Документ

Документом XML называется объект самого высокого уровня, который охватывает все представленное с его помощью содержимое, от первого символа до последнего. Если речь идет о документе XML, то под этим подразумевается и структура, и информационное наполнение данного конкретного документа XML.

Объявление документа

В определении требований к структуре документа XML (в спецификации языка XML) указано, что **объявление документа** является необязательным, но на практике его всегда следует включать в документ. Если в документе предусмотрено использование объявления, то это объявление должно находиться в документе на самом первом месте. В тексте документа ничто не должно предшествовать объявлению, даже пробельные символы (пробелы, символы возврата каретки, символы табуляции или что-либо еще), иными словами, не должно быть ничего.

Объявление представляет собой специальный дескриптор, который начинается с вопросительного знака (указывающего на то, что этот дескриптор является директивой препроцессора), за которым следует ключевое слово `xml`:

```
<?xml version="1.0"?>
```

В этом объявлении имеется один обязательный атрибут (напомним, что атрибут – это средство дополнительного описания элемента) – **атрибут version**. В предыдущем примере приведено объявление документа XML, а также указано, что он соответствует версии 1.0 спецификации XML.

По желанию в объявление можно включить еще один дополнительный атрибут – **атрибут encoding** с обозначением кодировки. Этот атрибут содержит сокращенное обозначение типа символьного набора, который используется в документе XML. Документы XML могут быть созданы с применением нескольких разных символьных наборов, наиболее известными из которых являются UTF-16 и UTF-8. Символьный набор UTF-16 по существу соответствует спецификации Unicode, которая представляет собой спецификацию кодировки, позволяющую представлять символы большинства языков, используемых в современном мире. По умолчанию применяется метод кодировки UTF-8, который обеспечивает обратную совместимость с более старыми версиями спецификации ASCII. Ниже приведен пример объявления, представленного в полной форме.

```
<?xml version='1.0' encoding='UTF-8'?>
```

В спецификации XML строго запрещено использование имен элементов, которые начинаются с букв `xml`; имена всех элементов с этим префиксом зарезервированы для дальнейшего расширения языка.

Элементы документа

Элементы обозначаются именами и служат для непосредственного представления информации, хранящейся в документе. С помощью элементов может быть выражена практически любая информация. Сам документ XML представляет собой линейную последовательность символов, а элементы своими начальными и конечными дескрипторами выделяют определенный фрагмент этой последовательности как принадлежащей исключительно им. Таким образом, обычно в состав элемента входит согласованная пара дескрипторов, состоящая из начального и конечного дескрипторов, которые называют также открывающим и закрывающим дескрипторами. Но предусмотрена также возможность при отсутствии информационного наполнения, заключенного между начальным и конечным дескрипторами, заменить эти два дескриптора одним, **содержащим признак закрытия в самом себе**. Это означает, что

элементы, не имеющие информационного наполнения, или **пустые элементы**, могут быть представлены с помощью любого из двух указанных способов.

По своей структуре элементы XML во многом напоминают дескрипторы HTML. Открывающий дескриптор начинается с открывающей угловой скобки (<), содержит имя, а также, возможно, несколько атрибутов, за которыми следует закрывающая угловая скобка (>):

```
<ATagForANormalElement >
```

Исключением из этого правила являются элементы, содержащие в себе признак закрытия; в этом случае закрывающей угловой скобке начального дескриптора предшествует символ /, а конечный дескриптор отсутствует:

```
<AselfClosingElement/>
```

Конечные дескрипторы содержат точно такое же имя, как и начальные дескрипторы (в котором также учитывается регистр), но в конечном дескрипторе перед именем элемента должна находиться косая черта (/).

```
<ATagForANormalElement >     <== Открывающий дескриптор
Some data or whatever can go in here.
We're still going strong with our data.
</ATagForANormalElement >    <== Закрывающий дескриптор
```

Элементы могут также включать атрибуты (о чем речь пойдет немного позже) в составе открывающего (но не закрывающего) дескриптора элемента. Наконец, элементы могут включать другие элементы, но в таком случае внутренний элемент должен быть закрыт прежде, чем будет закрыт внешний элемент:

```
<OuterElement>
   <InnerElement>
   </InnerElement>
</OuterElement>
```

Мы вернемся к описанию элементов после определения того, что подразумевается под термином "формально правильный".

Узлы

В связи с тем что элементы в документе XML должны подчиняться строгим правилам вложенности, между элементами естественным образом формируются отношения вложенности, которые образуют иерархическую структуру, имеющую древовидную форму (которая возникает почти во всех структурах, связанных иерархическими отношениями), показанную на рис. 16.2. Каждая конечная точка связи в этом дереве называется **узлом**.

Общепринятые способы обработки документов XML обеспечивают переход по его иерархической структуре с учетом того, на каком уровне находятся требуемые узлы, а также выборку значений атрибутов и элементов, соответствующих конкретному узлу.

Корневой узел

Изучая структуру документов XML, важно помнить, что каждый документ, представленный в виде иерархии узлов, должен иметь так называемый **корневой узел**. Следует еще раз подчеркнуть, что каждый документ XML имеет один (и только один)

корневой узел. Корневой узел представляет собой элемент, который содержит все прочие элементы в документе, в том числе элемент самого документа. Корневой узел может рассматриваться как связующая точка, с которой прямо или косвенно соединяются все нижележащие узлы, образующие структуру, в которую укладывается все содержимое любого конкретного документа XML. Итак, каждый документ имеет единственный корневой узел, и ему присваивается определенное имя (о чем речь пойдет немного позже).

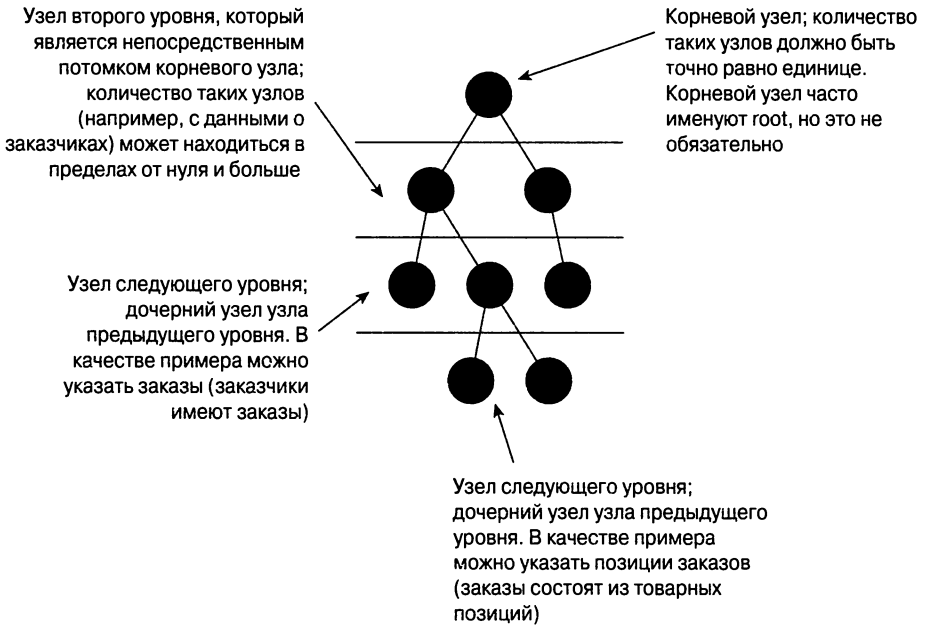


Рис. 16.2. Иерархическая структура документа XML

Корневой узел документа не имеет какого-то закрепленного за ним имени, допустим, `root` (корневой). Безусловно, иногда встречаются документы XML, в которых действительно имеется корневой узел с именем `root` (либо `Root`, либо `ROOT`, и т.д.). Но фактически имена корневых узлов должны соответствовать точно такой же схеме именования, как и имена любых других узлов (представленных в виде элементов), за одним исключением — имя корневого узла должно быть уникальным во всем документе. Иными словами, больше ни один элемент во всем документе не может иметь такое же имя, как и корневой узел.

Документ XML и фрагмент XML (соответствующий поддереву иерархического дерева документа) принципиально отличаются тем, что документ XML имеет корневой узел. Выполнение операций выборки данных с помощью СУБД SQL Server часто приводит к получению небольших отрывков кода XML, принадлежащих к более крупному целому документу. Такие данные (имеющие строковый формат) и представляют собой фрагменты XML. Но ни один фрагмент XML не может рассматриваться как равнозначный целому документу, поскольку в нем отсутствует корневой узел.

Атрибуты

Атрибуты могут быть представлены только в контексте элемента. Атрибуты применяются как средство дополнительного описания элемента и находятся в пределах границ открывающего дескриптора элемента:

```
<SomeElement MyFirstAttribute="Hi There" MySecondAttribute="25">
    Optionally, some other XML
</SomeElement>
```

Независимо от того, к какому типу данных относится информация, представленная в виде значения атрибута, это значение должно быть заключено в парные одинарные или двойные кавычки.

По умолчанию документы XML не поддерживают концепцию типа данных. Тем не менее предусмотрены некоторые способы формулировки правил, которые определяют требования по применению отдельных структурных компонентов документа, о чем пойдет речь ниже в данной главе. В дальнейшем будет показано, что эти способы позволяют обеспечить применение требуемых типов данных в соответствии с заданными правилами, но в самом языке XML эти правила не предусмотрены.

Формально правильный документ как документ, не имеющий дефектов структуры

Основным требованием к любому документу XML является то, чтобы он соответствовал определению формально правильного документа. Определение формально правильного документа XML регламентирует, какие элементы могут входить в состав документа, какую структуру он должен иметь и из каких частей состоять.

Понятие формально правильного документа распространяется на документы, формируемые с использованием любого языка, основанного на SGML. В определенной степени требования по обеспечению формальной правильности распространяются даже на документы HTML, хотя и соблюдаются менее строго, поскольку в самом языке HTML изначально допускалась большая свобода выбора способа представления информации, а браузеры автоматически исправляют или игнорируют многие ошибки в обрабатываемых ими документах.

Несмотря на то что HTML относится к категории языков, основанных на дескрипторах, документы, сформированные с помощью этого языка, часто имеют довольно рыхлую структуру. С другой стороны, в языке XML предусмотрены гораздо более строгие правила, касающиеся того, какие конструкции являются допустимыми. Ниже приведена краткая сводка этих правил.

- Каждый документ XML должен иметь уникальный корневой узел.
- Каждому открывающему дескриптору должен соответствовать закрывающий дескриптор с таким же именем (в котором учитывается регистр); закрывающий дескриптор может отсутствовать, только если открывающий дескриптор содержит признак закрытия в самом себе.
- Не допускается, чтобы за открывающим дескриптором одного элемента непосредственно следовал закрывающий дескриптор другого элемента.
- В информационном наполнении элемента не допускается непосредственно использовать символы, которые предназначены для разметки структуры документа XML и его интерпретации с помощью синтаксического анализатора

XML. Если необходимо представить любой из таких специальных символов, то вместо него должна использоваться управляющая последовательность (которая снова преобразуется в исходный символ во время выборки информационного наполнения элемента).

Ниже приведен пример формально правильного документа.

```
<?xml version="1.0" encoding="UTF-8"?>
<ThisCouldBeCalledAnything>
  <AnElement>
    <AnotherElement AnAttribute="Some Value">
      <AselfClosingElement AnAttributeThatNeedsASpecialCharacter=
"Fred&quot;s flicks"/>
    </AnotherElement>
  </AnElement>
</ThisCouldBeCalledAnything>
```

Обратите внимание на то, что объявление документа, находящееся в первой строке, не требует применения закрывающего дескриптора. Это связано с тем, что объявление представляет собой не элемент, а директиву препроцессора. По существу, с помощью этой директивы синтаксическому анализатору XML передается информация, необходимая для осуществления реальных действий по обработке документа XML.

Приведенное выше изложение требований к формально правильному документу XML является чрезвычайно кратким, но содержит основные сведения по этой теме, учитывая то, что в данной книге для описания тематики XML отведено мало места.

Понимание этих концепций становится крайне важным при изучении следующей главы. Приведенный ниже пример должен многое разъяснить, но если читатель после изучения этого кода XML будет чувствовать себя неуверенно, рекомендуем еще раз внимательно прочитать изложенное выше, а также ознакомиться с упомянутой книгой, Professional XML, или какой-то другой книгой по языку XML. От успешного освоения изложенного здесь материала зависит то, насколько продуктивным станет изучение вами проблематики применения стилей и схем в конце данной главы, не говоря уже о следующей главе.

Пример кода XML

В настоящей главе, как и при описании любых других тем, приведено несколько примеров. Как уже было сказано, данная книга не посвящена описанию языка XML, поэтому количество приведенных примеров будет не столь велико, но по крайней мере они вполне позволят проиллюстрировать сказанное.

Кроме того, в этой и следующей главах будет показано, насколько упрощается работа, если используются инструментальные средства редактирования XML определенного типа. Безусловно, язык XML полностью основан на применении чисто текстовой информации, поэтому любой документ XML может быть открыт и отредактирован даже в редакторе Notepad, но проблема заключается в том, что с помощью этого редактора не удастся выполнить проверку на наличие ошибок. Прежде всего необходимо определить, является ли документ формально правильным. Если документ состоит из нескольких строк, то, разумеется, его можно легко проверить визуально, но если документ очень большой или к нему прилагается определение XSL, то проверка документа значительно усложняется.

Отметим, что можно выполнить довольно успешную предварительную проверку того, является ли документ XML формально правильным, открыв его в браузере Internet Explorer корпорации Microsoft; если данный документ не является формально правильным, программа сразу же об этом сообщит.

В качестве примера в данном разделе рассмотрим представление некоторых данных из базы данных `Northwind` в коде XML. В этом случае речь идет о некоторой информации заказов. Для этого начнем с нескольких простых элементов и будем постепенно наращивать сложность представления.

Прежде всего напомним, что для каждого создаваемого документа XML необходимо предусмотреть корневой узел. В частности, отметим, что корневой узел может иметь любое имя, при условии, что элемент с этим именем будет уникальным во всем документе. Чаще всего такая задача решается таким образом, что корневому узлу присваивается имя `root`. Еще один часто применяемый способ состоит в том, что элементу корневого узла присваивается имя, характеризующее весь данный конкретный документ XML в целом.

В рассматриваемом примере начнем с рассмотрения чрезвычайно простого документа, поэтому назовем элемент корневого узла просто `root`:

```
<root>
</root>
```

Итак, этого нам оказалось достаточно, чтобы создать свой первый формально правильный документ XML. Обратите внимание на то, что он не содержит дескриптор `<?xml ... ?>`, который был показан в предыдущем примере. Безусловно, этот дескриптор можно было применить и в данном примере, но фактически он является необязательным. Но с использованием этого дескриптора связано определенное условие – если он предусмотрен в документе, то должен находиться на первом месте. Таким образом, учитывая рекомендации по разработке документов XML, а также стремясь создать предельно понятный код, исправим недостаток предыдущего варианта кода и введем указанный дескриптор:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
</root>
```

Фактически согласно определению языка XML любой дескриптор, начинающийся с подстроки `<?xml`, рассматривается как зарезервированный; еще одно требование состоит в том, что в именах создаваемых разработчиком дескрипторов не должна встречаться подстрока `xml`, поскольку она зарезервирована консорциумом W3C для использования в настоящее время (или в будущем, когда появятся следующие версии XML).

Итак, мы подготовили две версии своего первого формально правильного документа XML. К сожалению, этот документ в своем текущем виде является предельно простым, ведь он фактически не несет в себе никакой информации. А поскольку в рассматриваемом примере требуется представить информацию заказа, приступим к сбору информации, позволяющей описать содержимое заказа. Вначале введем дескриптор `Order`:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Order/>
</root>
```

Очевидно, что документ уже приобрел какой-то смысл. Взглянув на этот документ XML, можно утверждать, что в нем представлен один заказ (элемент `Order`), но об этом заказе все еще ничего не известно. Дополним элемент, введя в него несколько атрибутов:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Order CustomerID="ALFKI" OrderID="10643" OrderDate="1997-08-25T00:00:00" />
</root>
```

Таким образом, очевидно, что даже после такого небольшого дополнения формируется легко доступный для понимания документ, который содержит описанные ниже сведения.

- Компания заказчика имеет идентификационный номер `ALFKI`.
- Заказу присвоен идентификационный номер `10643`.
- Заказ был размещен 25 августа 1997 года.

По существу, представленная здесь информация соответствует содержимому одной строки из таблицы `Orders` базы данных `Northwind`, входящей в состав программного обеспечения SQL Server. А несколько заказов одного и того же заказчика могут быть представлены примерно так:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Order CustomerID="ALFKI" OrderID="10643" OrderDate="1997-08-25T00:00:00" />
  <Order CustomerID="ALFKI" OrderID="10692" OrderDate="1997-10-03T00:00:00" />
  <Order CustomerID="ALFKI" OrderID="10702" OrderDate="1997-10-13T00:00:00" />
  <Order CustomerID="ALFKI" OrderID="10835" OrderDate="1998-01-15T00:00:00" />
</root>
```

Безусловно, в рассматриваемом примере представлен полностью допустимый и даже формально правильный документ XML, но фактически язык XML позволяет представлять иерархические данные, а таких данных в рассматриваемом примере не наблюдается. В частности, было бы удобно применить немного другую структуру документа XML и отразить в нем то представление, что заказчики (рассматриваемые как объекты) обычно находятся на более высоком уровне иерархии объектов (объекты заказчиков можно считать родительскими по отношению к объектам заказов). Такую структуру можно представить, изменив способ представления данных о заказчиках. В предыдущем примере данные о заказчике были отображены с помощью атрибута, но существует возможность предусмотреть для представления этих данных полностью самостоятельный элемент (в том числе имеющий собственные атрибуты) и вложить элементы с описанием заказов данного конкретного заказа в элемент, представляющий самого заказчика:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Order OrderID="10643" OrderDate="1997-08-25T00:00:00" />
    <Order OrderID="10692" OrderDate="1997-10-03T00:00:00" />
    <Order OrderID="10702" OrderDate="1997-10-13T00:00:00" />
    <Order OrderID="10835" OrderDate="1998-01-15T00:00:00" />
  </Customer>
</root>
```

Если же заказчиков больше одного, то не возникает никаких проблем – достаточно ввести дополнительные элементы с описаниями других заказчиков:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
    <Order OrderID="10643" OrderDate="1997-08-25T00:00:00" />
    <Order OrderID="10692" OrderDate="1997-10-03T00:00:00" />
    <Order OrderID="10702" OrderDate="1997-10-13T00:00:00" />
    <Order OrderID="10835" OrderDate="1998-01-15T00:00:00" />
  </Customer>
  <Customer CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria">
    <Order OrderID="10365" OrderDate="1996-11-27T00:00:00" />
    <Order OrderID="10507" OrderDate="1997-04-15T00:00:00" />
    <Order OrderID="10535" OrderDate="1997-05-13T00:00:00" />
    <Order OrderID="10573" OrderDate="1997-06-19T00:00:00" />
  </Customer>
</root>
```

Безусловно, допустимое количество уровней иерархии ничем не ограничивается (разумеется, не считая того, что такие ограничения могут зависеть от используемой программы синтаксического анализатора). Например, ничто не мешает нам ввести в структуру заказа отдельные элементы, представляющие в заказе товарные позиции.

Сравнение способов представления данных с помощью элементов и атрибутов

Прежде всего следует отметить, что нет твердых и окончательно установленных правил в отношении того, какую информацию следует представлять с помощью элементов или атрибутов. Атрибут описывает то, что является свойством элемента, к которому относится сам атрибут. Дочерние элементы (или, в древовидном представлении, дочерние узлы) родительского элемента имеют в основном такое же назначение. Как же в таком случае принять решение об использовании тех или других? Для чего вообще нужны атрибуты? Отметим, что в данном случае, как и во многих других, все зависит от дополнительных обстоятельств.

Применение атрибутов оправдано в тех ситуациях, когда некоторое значение находится во взаимно однозначной связи (связи "один к одному") с элементом и является его неотъемлемой частью. Например, в базе данных `Northwind` с каждым идентификатором заказчика связано только одно название компании, а этот случай является идеальным с точки зрения использования атрибута. По мере преобразования реляционной структуры данных в иерархическую структуру XML часто обнаруживается, что столбцы таблицы становятся приемлемыми кандидатами для применения в качестве атрибутов элемента, непосредственно относящихся к отдельным строкам таблицы.

Использование элементов в большей степени оправдано, если между самим элементом и теми данными, которые он описывает, чаще всего проявляется связь "один ко многим". Согласно условиям примера, приведенного выше в данной главе, каждый заказчик может иметь много заказов. С формальной точки зрения можно было бы представить каждый заказ как атрибут элемента, относящегося к заказчику, но в таком случае потребовалось бы снова и снова повторять значительную часть информации элемента, касающейся заказчика. Аналогичным образом, если бы в базе данных `Northwind` была предусмотрена возможность, чтобы заказчики имели псевдонимы

(т.е. чтобы компании одних и тех же заказчиков выступали под разными названиями), то потребовалось бы включить такой элемент, как "name", дочерний по отношению к элементу заказчика, и предусмотреть использование атрибутов элемента "name" для описания отдельных вариантов названий одной и той же компании.

Но независимо от принятого подхода, рекомендуем придерживаться правила, важность которого неоднократно подчеркивалась в данной книге, — учитывать необходимость соблюдать единообразный подход. Если какой-то фрагмент информации определен как предназначенный для представления в виде атрибута в одном месте, то следует неизменно продолжать представлять его в виде атрибута и в других частях общей структуры документа, при условии, что характер его использования в других частях не изменяется коренным образом. Иными словами, еще раз рекомендуем придерживаться единообразия.

Пространства имен

Очевидно, что язык XML предоставляет неограниченную свободу и позволяет создавать собственные дескрипторы, смешивать и согласовывать данные из разных источников и даже просто творчески подходить к созданию структуры документов, но из этого следует, что неизбежны определенные коллизии, в частности, связанные с тем, какие имена используются для обозначения элементов и атрибутов в разных частях структуры документа. Например, предположим, что в приложении, созданном для представления документов из исторической библиотеки, предусмотрен элемент "letter", который обладает вполне определенной структурой, подчиняется конкретным правилам и имеет определенный смысл (допустим, представляет собой описание письма, написанного одним историческим лицом другому), а в другом приложении элемент "letter", предположим, служит частью описания шрифтов (в частности, представляет связь между одним символом из набора символов и серией глифов).

Чтобы еще более наглядно подчеркнуть важность рассматриваемой темы, отметим, что специфика применения языка XML требует, чтобы различные отраслевые организации во всем мире постепенно пришли к соглашению об использовании единых принципов именования и представления структуры различной информации, обрабатываемой в соответствующих отраслях. Например, организациям, занимающимся комплектованием библиотек, приходится согласовывать друг с другом форматы элементов с описанием книг, пьес, кинофильмов, писем, эссе и т.д. С другой стороны, компании, занимающиеся разработкой операционных систем и (или) графических программ, согласовывают форматы элементов с описанием изображений, шрифтов и компоновок документов.

А теперь допустим, что простые, рядовые разработчики получили задание написать приложение, позволяющее представить содержимое библиотеки. Очевидно, что при описании книг и документов, представленных в библиотеке, часто приходится указывать, какие в них используются шрифты, и даже упоминать отдельные символы этих шрифтов, если они имеют особое значение. Таким образом, в разрабатываемых определениях документов XML обязательно придется применить для обозначения символов элементы с такими именами, как "letter", но не исключено, что в том же приложении, в котором упоминается шрифт, может идти речь о письме, написанным одним лицом другому (скажем, о письме Томаса Джефферсона Джорджу Вашингтону), и для этого придется применить элемент, который также имеет имя "letter", но в этом контексте означает письмо. Таким образом, возникает конфликт имен и нужен способ его разрешения.

Именно для этой цели предусмотрены **пространства имен**. Пространство имен описывает область определения элементов и атрибутов, а также регламентирует их структуру. Таким образом, структура, в рамках которой могут быть представлены письма в библиотеках (содержащая элементы "letter"), будет описана в пространстве имен, предусмотренном для библиотек. Аналогичным образом, компании, занимающиеся производством книжной продукции, могут предусмотреть собственные пространства имен для описания символов (также присвоив соответствующим элементам имя "letter"), поскольку для них наиболее важным является то, из каких символов состоят шрифты, применяемые в данной отрасли. Информация, относящаяся к пространству имен, хранится в справочном документе и может быть получена с использованием универсального идентификатора ресурсов (Uniform Resource Identifier — URI). Идентификаторы URI — это особые имена, близкие по своему назначению к URL, которые в конечном итоге позволяют получить необходимый справочный документ с информацией о пространстве имен.

Поэтому если потребуется создать такой документ XML, в котором используются элементы с одинаковыми именами, в данном случае "letter", но которые относятся к разным областям человеческой деятельности (в описанном случае к изучению писем исторических деятелей и представлению символов шрифтов), то достаточно указать, к каким пространствам имен относятся и те и другие элементы. Кроме того, необходимо ввести дополнительные уточнения в имена элементов и атрибутов, позволяющие подчеркнуть их специфику, представленную в определении пространства имен. Для этого предусмотрен способ уточнения имен элементов и атрибутов с указанием пространств имен, который позволяет обеспечить возможность ссылаться на различные части документа, сохраняя полную уверенность в том, что при этом не упоминается неправильный элемент или атрибут, даже если в документе имеются принципиально различные элементы и атрибуты, имеющие тем не менее одинаковые имена.

Чтобы связать с определенным пространством имен весь документ, достаточно ввести в определение корневого элемента документа ссылку на пространство имен в виде специального атрибута (именуемого `xmlns`). В этой ссылке должно быть указано и локальное имя (применяемое в документе для указания пространства имен), и идентификатор URI, который в конечном итоге позволяет обратиться к используемому справочному документу. Кроме того, ссылки на пространства имен (опять-таки формируемые с помощью атрибута `xmlns`) могут быть введены и в другие элементы документа, если требуется распространить действие данного конкретного пространства имен только на ту часть документа, которую занимает данный элемент.

Ниже приведен пример документа XML (с формальной точки зрения этот документ представляет собой так называемую *схему*), который будет использоваться в следующей главе. Этот документ имеет несколько особенностей, которые связаны со спецификой применения пространств имен, в частности, следующие особенности.

В документе имеются ссылки на три пространства имен. Одно из них относится к определению XDR (и действительно представляет собой документ XDR), другое представляет собой пространство имен типов данных Microsoft (в этом пространстве имен представлен перечень, определяющий количество и специфику различных типов данных) и, наконец, предусмотрено специальное пространство имен SQL, предназначенное для работы со средствами интеграции XML программы SQL Server.

Некоторые атрибуты (включая заданные в корневом элементе) уточняются с помощью информации о пространстве имен (в качестве примера можно указать атрибут `sql:relation`).

```
<?xml version="1.0" encoding="UTF-8"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
            xmlns:dt="urn:schemas-microsoft-com:datatypes"
            xmlns:sql="urn:schemas-microsoft-com:xml-sql"
            sql:xsl='../Customers.xsl'>
 <ElementType name="Root" content="empty"  />
 <ElementType name="Customers" sql:relation="Customers">
      <AttributeType name="CustomerID"/>
      <AttributeType name="CompanyName"/>
      <AttributeType name="Address"/>
      <AttributeType name="City"/>
      <AttributeType name="Region"/>
      <AttributeType name="PostalCode"/>
      <attribute type="CustomerID" sql:field="CustomerID"/>
      <attribute type="CompanyName" sql:field="CompanyName"/>
      <attribute type="Address" sql:field="Address"/>
      <attribute type="City" sql:field="City"/>
      <attribute type="Region" sql:field="Region"/>
      <attribute type="PostalCode" sql:field="PostalCode"/>
 </ElementType>
</Schema>
```

Тип данных `sql` предусматривает использование ссылок на несколько специальных атрибутов. Применяя этот тип данных, можно не задумываться над тем, предусмотрен ли также в пространстве имен типов данных Microsoft такой тип данных, как `field` или `relation`, поскольку коллизия имен исключена благодаря использованию полностью уточненных имен атрибутов. Даже если бы в пространстве имен типов данных был определен атрибут `field`, синтаксический анализатор XML все равно обрабатывал бы данный элемент согласно правилам, определенным в пространстве имен `sql`.

Содержимое элемента

Еще одной особенностью документов и элементов XML, заслуживающей упоминания (поскольку эта особенность непосредственно относится к излагаемой теме), является понятие содержимого элемента.

Элементы могут содержать данные, относящиеся не только к атрибутам и вложенным элементам, но и представляющие собой содержимое самого элемента. Безусловно, в определенном смысле часть содержимого элемента составляют вложенные в него элементы (вложенным элементом называется элемент, содержащийся в другом элементе), но язык XML допускает также, чтобы строковая текстовая информация содержалась непосредственно в элементе. Например, вполне допускается создание такого документа XML, который может выглядеть, в частности, следующим образом:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
   <Note Date="1997-08-25T00:00:00">
```

```
    The customer called in today and placed another order. Says they really
  like our work and would like it if we would consider establishing a location
  closer to their base of operations.
    </Note>
    <Note Date="1997-08-26T00:00:00">
    Followed up with the customer on new location. Customer agrees to
  guarantee us $5,000 per month in business to help support a new office.
    </Note>
   </Customer>
  </root>
```

Такое содержимое элементов `Note`, как "`The customer called ...`", не является ни отдельным элементом, ни атрибутом, но представляет собой допустимые данные XML.

Однако следует учитывать, что, несмотря на возможность представления подобных данных в языке XML, программные средства SQL Server не обеспечивают вывод данных непосредственно в этом формате. Дело в том, что в системах реляционных СУБД для представления данных применяются строки и столбцы, а этот формат представления является более строгим по сравнению с элементами и атрибутами. Для вывода данных, подобных приведенным выше примечаниям в элементах `Note`, необходимо преобразовать данные, полученные из базы данных, в новый формат. Некоторые способы преобразования данных рассматриваются в конце настоящей главы.

Применение схем и определений DTD для проверки допустимости формально правильных документов

К документам XML часто предъявляются требования, чтобы они были не только формально правильными, но и **допустимыми**. Допустимым считается документ XML, прошедший проверку с помощью другого документа, содержащего спецификацию, которая выражена в определенной форме. В настоящее время принято использовать документы со спецификациями, представленными только в двух формах — в виде определения типа документа (Document Type Definition — DTD) или схемы XML.

Документы этих двух типов, предназначенные для проверки допустимости других документов, выполняют в основном одни и те же функции. В стандарте языка XML определено, каким наиболее важным требованиям должен отвечать любой формально правильный документ XML, а определения DTD и схемы XML позволяют указать, каким правилам должны соответствовать документы XML, относящиеся к определенному классу. Определения DTD и схемы XML воплощают в себе два разных подхода к проверке допустимости документов XML и характеризуются своими преимуществами и недостатками, которые описаны ниже.

- Определения DTD. Методы проверки допустимости документов на основе определений DTD являются давно сложившимися, проверенными и надежными. Определения DTD впервые были введены в языке SGML (язык XML является подмножеством языка SGML; это означает, что SGML — надмножество XML, но чрезвычайно сложное в изучении) и имеют то преимущество, что на них основаны чрезвычайно распространенные и общепринятые методы организации работы. Уже заранее подготовлено огромное количество определений DTD для документов самых разных типов, поэтому часто бывает достаточно лишь выбрать требуемое определение и воспользоваться им.

- Недостатком определений DTD (а, как известно, недостатки можно найти во всем) является то, что в приведенном выше описании определений DTD как "давно сложившихся, проверенных и надежных" есть слово "давно". Не все, что появилось давно, можно назвать плохим, но в данном случае вполне очевидно, что скорость развития методов, основанных на использовании определений DTD, не соответствует тем стремительным изменениям, которые происходят в технологии обработки документов. Документы DTD фактически не позволяют реализовать даже такие, казалось бы, простейшие требования, как регламентация форматов типов данных.
- Схемы XML. Схемы XML имеют такое явное преимущество, как возможность строгого контроля типов. Но наиболее привлекательной особенностью схем XML является то, что они позволяют определять собственные **сложные типы данных**, т.е. типы, представляющие собой сочетание одного или нескольких других типов данных (включая другие сложные типы данных) или требующие применения специализированных средств сопоставления с шаблонами (например, номер карточки социального обеспечения – это просто число, но подчиняется особым правилам форматирования, соблюдение которых можно легко проверить с помощью схемы XML). Кроме того, как показывает само название схем XML, они являются документами XML, и в этом также состоит их преимущество. В частности, из этого следует, что все свои навыки создания документов XML разработчики могут также применить для создания схем (хотя для этого все же потребуется еще многое узнать). Кроме того, и схемы могут быть разработаны как описывающие сами себя, вплоть до того, что может быть обеспечена возможность проверять допустимость самих схем по другим схемам.

Средства формирования документов XML, предусмотренные в СУБД SQL Server

В предыдущих разделах приведены основные сведения о языке XML, а в данном разделе рассматриваются средства поддержки XML, предусмотренные в СУБД SQL Server.

Функциональные средства XML были введены в состав программного обеспечения SQL Server сравнительно недавно. В действительности возможность работы с языком XML впервые стала предоставляться в версии SQL Server 7.0 с помощью отдельно загружаемого дополнения. Более того, значительная часть этих функциональных средств в большей степени представляла собой дополнение к серверу IIS (Internet Information Server), чем к СУБД SQL Server.

Начиная с версии SQL Server 2000 та часть программного обеспечения, которая касалась поддержки XML, была перенесена в программный компонент, который был назван корпорацией Microsoft моделью "Web Release", после чего вышло несколько обновляемых выпусков этого программного обеспечения. В настоящее время с выходом версии SQL Server 2005 средства поддержки языка XML были окончательно перенесены в основной состав программного продукта. Безусловно, все введенные ранее функциональные возможности продолжают поддерживаться, но в версии SQL Server 2005 предусмотрен более централизованный набор средств, благодаря которому поддержка языка XML становится неотъемлемой частью функционирования системы, а

не вспомогательным средством, каким в свое время (при подготовке предыдущих выпусков) считался язык XML.

В версии SQL Server 2005 функциональные средства поддержки языка XML выходят на передний план в нескольких описанных ниже направлениях эксплуатации.

- Поддержка нескольких методов выборки данных из обычных столбцов и оформление этих данных в формате XML.
- Обеспечение возможности непосредственного хранения данных XML в СУБД SQL Server.
- Предоставление функций запроса к данным, хранящимся в первоначальном формате XML, с помощью языка XQuery.
- Поддержка средства обеспечения целостности данных, хранящихся в формате XML, с помощью схем XML.
- Обеспечение индексации данных XML.

Следует отметить, что здесь перечислены лишь основные направления поддержки XML.

Для обеспечения перечисленных выше возможностей часто приходится использовать несколько функциональных областей поддержки XML, поэтому рассмотрим последовательно каждое из направлений применения языка XML.

Выборка реляционных данных в формате XML

Это направление поддержки языка XML в СУБД SQL Server уже почти полностью сложилось еще до появления SQL Server 2005. Для этого были предусмотрены программные средства, имеющие целый ряд параметров, а сами эти параметры включали еще больше дополнительных параметров; в конечном итоге это позволяло создавать весьма разнообразные методы обработки данных. Ниже описаны некоторые программные конструкции применявшиеся с самого начала.

Конструкция FOR XML

Конструкция `FOR XML` является основой большей части различных доступных моделей внедрения языка XML. Если не считать схем отображения XML (весьма совершенных конструкций, которые будут кратко описаны ниже в данной главе) и методов применения языка XPath, конструкция `FOR XML` представляет собой основной способ оформления операторов, выполняемых в СУБД SQL Server, которые позволяют осуществлять выборку данных в виде кода XML, а не обычного результирующего набора. По существу, конструкция `FOR XML` определена как всего лишь еще одна необязательная конструкция, которая добавлена в конце существующей синтаксической структуры оператора `SELECT` языка T-SQL.

Еще раз рассмотрим синтаксис оператора `SELECT`, приведенный в главе 3:

```
SELECT <column list>
[FROM <source table(s)>]
[WHERE <restrictive condition>]
[GROUP BY <column name or expression using a column in the SELECT list>
[HAVING <restrictive condition based on the GROUP BY results>]
[ORDER BY <column list>]
[FOR XML {RAW|AUTO|EXPLICIT[, XMLDATA][, ELEMENTS][, BINARY base64]|PATH]
[OPTION (<query hint>, [, ...n])]
```

Читатель уже должен быть хорошо знаком с основной частью этого синтаксиса, поскольку все компоненты синтаксической структуры оператора SELECT рассматривались во всех предыдущих главах, а в этом разделе в основном сосредоточимся на изучении строки с ключевым словом FOR XML.

В конструкции FOR XML предусмотрено несколько различных начальных опций, позволяющих указать, как должны быть отформатированы результаты запроса в виде кода XML.

- Опция RAW. Ключевое слово RAW указывает, что каждая строка данных в результирующем наборе должна быть возвращена как один элемент данных. Элемент получает имя "row", а каждое поле строки оформляется в виде атрибута элемента "row". Даже в случае соединения нескольких таблиц с помощью опции RAW вывод результатов производится с таким же количеством элементов, которое соответствует количеству строк, возвращенных обычным запросом SQL.
- Опция AUTO. При использовании опции RAW каждый элемент получает обозначение, взятое либо из имени таблицы, либо из псевдонима имени таблицы, которая является источником данных. Если в результате запроса формируются данные, полученные больше чем из одной таблицы, то данные из каждой таблицы разбиваются на отдельные вложенные элементы. Кроме того, при использовании опции AUTO поддерживается также дополнительная опция ELEMENTS, позволяющая указать, что данные полей должны быть представлены в виде элементов, а не атрибутов.
- Опция EXPLICIT. Безусловно, опция EXPLICIT требует применения наиболее сложных синтаксических конструкций при оформлении запроса, но в конечном итоге позволяет получить наибольший контроль над тем, как будут выглядеть данные в коде XML. Кроме того, опция EXPLICIT позволяет отчасти определить иерархию возвращаемых данных, а затем сформировать запрос таким образом, чтобы каждый фрагмент полученных данных принадлежал к конкретному уровню иерархии (и обозначался соответствующим ему дескриптором) согласно сформулированным требованиям. Тем не менее возможности опции EXPLICIT главным образом перекрываются предусмотренными в опции PATH, поэтому поддержка опции EXPLICIT осуществляется главным образом для обеспечения обратной совместимости.
- Опция PATH. Поддержка ключевого слова PATH была впервые введена в версии SQL Server 2005 для обеспечения возможности достижения такого же разнообразия функциональных средств, которые предоставляет опция EXPLICIT, но в более удобной форме. Поэтому, чтобы добиться высокой степени контроля над форматом выходных данных, следует использовать опцию PATH.

Необходимо учитывать, что ни одна из указанных опций не обеспечивает формирования обязательного корневого элемента. Если требуется обеспечить, чтобы формируемые документы XML рассматривались как формально правильные, то необходимо ввести полученные результаты между открывающим и закрывающим дескрипторами корневого элемента должным образом или предоставить возможность выполнить эту задачу самой СУБД SQL Server (с помощью описанной ниже опции ROOT). Хотя на первый взгляд такая особенность применения опции FOR XML рассматривается как причина затруднений, в этом есть свои преимущества. Дело в том, что указанный подход позволяет формировать более сложные документы XML, объединяя результаты нескольких запросов XML в одну строку и в дальнейшем оформляя полученные результаты в виде одного документа XML.

Кроме описанных выше четырех основных опций форматирования, предусмотрено также пять других необязательных параметров, позволяющих дополнительно уточнять формат выходных данных, предоставляемых в СУБД SQL Server при выполнении запроса с ключевым словом FOR XML.

- Опция XMLDATA. Эта опция служит для СУБД SQL Server указанием на то, что формируемым результатам должно предшествовать определение схемы XML. Схема определяет структуру документа (в том числе состав используемых типов данных), а также правила, которым должны соответствовать данные XML. Следует учитывать, что используемая при этом схема не полностью соответствует тому определению, которое было предложено в спецификации консорциума W3C ко времени написания настоящей книги.
- Опция ELEMENTS. Эта опция является применимой, только если используется также опция форматирования AUTO. Опция ELEMENTS служит для СУБД SQL Server указанием на то, что данные полей в возвращаемых результатах должны быть оформлены в виде вложенных элементов, а не атрибутов.
- Опция BINARY BASE64. Эта опция является для СУБД SQL Server указанием на то, что содержимое всех полей с двоичными данными (binary, varbinary, image) должно быть закодировано в формате base64. Если используется опция AUTO, то опция BINARY BASE64 вводится в действие по умолчанию (т.е. применяется в СУБД SQL Server, даже если она явно не задана). А при использовании опции EXPLICIT или RAW ключевое слово BINARY BASE64 не вводится в действие по умолчанию, но в настоящее время является единственно возможным вариантом получения приемлемых данных. На данный момент корпорация Microsoft осуществляет план, реализация которого в конечном итоге приведет к тому, что при использовании опции EXPLICIT или RAW будет формироваться ссылка URL на соответствующие двоичные данные (если не будет указано, что для них применяется кодировка BASE64), но этот план еще не реализован.
- Опция TYPE. Эта опция рассматривается в СУБД SQL Server в качестве указания на то, что возвращаемые результаты должны быть обозначены как относящиеся к типу данных xml, а не к применяемому по умолчанию символьному типу данных Unicode.
- Опция ROOT. С помощью данной опции можно указать, что корневой элемент должен быть сформирован в СУБД SQL Server, поэтому разработчику не нужно об этом заботиться. В таком случае можно либо явно задать имя корневого элемента, либо предусмотреть использование имени корневого элемента, заданного по умолчанию (root).

Рассмотрим описанные выше опции более подробно.

Опция RAW

Опция RAW относится к числу наиболее простых. В основе ее применения лежит такая идея, что в СУБД уже есть все, что необходимо для обработки содержащихся в ней данных, поэтому можно обойтись без какого-либо специального форматирования и достаточно ввести лишь абсолютный минимум описательной информации, чтобы преобразовать строку реляционных данных в элемент, содержащий данные XML. Сам элемент получает имя "row" (об этом следует помнить, оформляя данные из разных таблиц), а имя каждого столбца в списке выборки используется в качестве

имени атрибута. При этом применяется такое имя, которое приобрел бы столбец после оформления содержащихся в нем данных в виде результирующего набора, если бы в рассматриваемом операторе SELECT не использовалась конструкция FOR XML.

Одним из недостатков такого способа именования атрибутов является то, что необходимо обеспечить присваивание имени каждому столбцу. При обычных условиях, если выполняется агрегирование или формируется другой вычисленный столбец и не предоставляется псевдоним, то СУБД SQL Server просто не показывает в выходных данных заголовок для этого столбца, а при выполнении запросов с конструкцией FOR XML каждый столбец обязательно должен иметь имя, поэтому не забывайте обозначать вычисленные и прочие безымянные столбцы псевдонимами.

Итак, начнем рассмотрение данной темы с относительно простого примера. Предположим, что руководитель поручил нам подготовить запрос для получения списка заказов нескольких заказчиков, скажем, с такими идентификаторами CustomerID, как ALFKI и ANTON. Безусловно, материал, изложенный в предыдущих главах, позволяет легко справиться с указанной задачей, поэтому составим примерно такой запрос:

```
USE Northwind
SELECT Customers.CustomerID,
  Customers.CompanyName,
  Orders.OrderID,
  Orders.OrderDate
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
```

После выполнения этого запроса будут получены следующие результаты:

```
ANTON Antonio Moreno Taqueria        10365   1996-11-27 00:00:00.000
ANTON Antonio Moreno Taqueria        10507   1997-04-15 00:00:00.000
...
...
ALFKI Alfreds Futterkiste           11081   2000-07-22 00:00:00.000
ALFKI Alfreds Futterkiste           11087   2000-08-05 17:37:52.520
```

Но оказалось, что теперь требуется представить эти результаты в виде кода XML. Сведения, изложенные выше в данной главе, позволяют легко справиться с поставленной задачей, поэтому просто добавим к запросу два ключевых слова, FOR XML и RAW:

```
USE Northwind
SELECT Customers.CustomerID,
  Customers.CompanyName,
  Orders.OrderID,
  Orders.OrderDate
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
FOR XML RAW
```

На данный момент полученные результаты вполне отвечают поставленным требованиям. Эти выходные данные характеризуются взаимно однозначным соответствием с теми данными, которые находятся в результирующем наборе, полученном с помощью стандартного запроса SQL:

```
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria"
OrderID="10365" OrderDate="1996-11-27T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria"
OrderID="10507" OrderDate="1997-04-15T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria"
OrderID="10535" OrderDate="1997-05-13T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria"
OrderID="10573" OrderDate="1997-06-19T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10643"
OrderDate="1997-08-25T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria"
OrderID="10677" OrderDate="1997-09-22T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria"
OrderID="10682" OrderDate="1997-09-25T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10692"
OrderDate="1997-10-03T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10702"
OrderDate="1997-10-13T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10835"
OrderDate="1998-01-15T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria"
OrderID="10856" OrderDate="1998-01-28T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10952"
OrderDate="1998-03-16T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="11011"
OrderDate="1998-04-09T00:00:00" />
```

Следует учитывать, что во время работы программы Management Studio происходит усечение значений любых полей, длина которых превышает величину, заданную в меню Options *на вкладке* Results to Text *(этот максимум равен 8192). Такая проблема обнаруживается в окне результатов (при отображении в виде сетки или текста), а также при выводе данных непосредственно в файл. Причиной указанного нарушения в работе является само инструментальное средство, а не СУБД SQL Server. При использовании для выборки результатов другого метода (например, интерфейса ADO) подобная проблема не возникает.*

Итак, сформировано по одному элементу XML в расчете на каждую строку данных полученного результирующего набора. Информация из всех столбцов, независимо от того, какая таблица была источником данных, представлена в виде атрибутов элемента "`row`". К недостаткам этого способа представления данных относится то, что с его помощью невозможно представить подлинный иерархический характер данных, поскольку (в данном случае) допускается лишь возможность показать, какие заказы разместили заказчики. С другой стороны, если для обработки полученного кода XML применяется модель DOM, то достигается определенное преимущество, поскольку количество уровней иерархии представления документа остается весьма небольшим, поэтому, в зависимости от конкретных условий, можно добиться некоторого уменьшения потребности в оперативной памяти и повышения производительности.

Опция AUTO

Применение опции AUTO связано с несколько иным подходом к обработке данных по сравнению с опцией RAW. При использовании опции AUTO предпринимается попытка создания документа XML, имеющего немного более приемлемую структуру с точки зрения пользователя, поскольку имена элементам присваиваются с учетом того, какое имя имеет сама таблица (или с учетом псевдонима таблицы, если он предусмотрен). Кроме того, средства поддержки опции AUTO действуют с учетом той предпосылки, что данные, возможно, организованы на основе какой-то основополагающей иерархической структуры, которая, скорее всего, должна быть представлена и в документе XML.

Вернемся к примеру с заказами заказчиков, который рассматривался в предыдущем разделе, но на этот раз воспользуемся опцией AUTO, чтобы узнать, чем отличаются полученные результаты от тех довольно простых результатов, которые были получены с помощью опции RAW:

```
USE Northwind
SELECT Customers.CustomerID,
Customers.CompanyName,
  Orders.OrderID,
  Orders.OrderDate
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
FOR XML AUTO
```

Первое очевидное различие состоит в том, что вместо имени элемента "raw" применяется имя или псевдоним таблицы, которая является источником данных, но еще более важное различие обнаруживается после более внимательного рассмотрения кода XML (автор немного отформатировал полученные результаты, чтобы они стали более удобными для чтения):

```
<Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria">
  <Orders OrderID="10365" OrderDate="1996-11-27T00:00:00" />
  <Orders OrderID="10507" OrderDate="1997-04-15T00:00:00" />
  <Orders OrderID="10535" OrderDate="1997-05-13T00:00:00" />
  <Orders OrderID="10573" OrderDate="1997-06-19T00:00:00" />
</Customers>
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Orders OrderID="10643" OrderDate="1997-08-25T00:00:00" />
</Customers>
<Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria">
  <Orders OrderID="10677" OrderDate="1997-09-22T00:00:00" />
  <Orders OrderID="10682" OrderDate="1997-09-25T00:00:00" />
</Customers>
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Orders OrderID="10692" OrderDate="1997-10-03T00:00:00" />
  <Orders OrderID="10702" OrderDate="1997-10-13T00:00:00" />
  <Orders OrderID="10835" OrderDate="1998-01-15T00:00:00" />
</Customers>
<Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria">
  <Orders OrderID="10856" OrderDate="1998-01-28T00:00:00" />
</Customers>
```

```
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Orders OrderID="10952" OrderDate="1998-03-16T00:00:00" />
  <Orders OrderID="11011" OrderDate="1998-04-09T00:00:00" />
</Customers>
```

Данные, источником которых является вторая таблица (в соответствии с тем, что указано в списке выборки), вкладываются в данные, источником которых является первая таблица. В этом случае происходит вложение элементов `Orders` в элементы `Customers`. А если бы в списке выборки был в первую очередь приведен столбец из таблицы `Orders`, то элементы `Customers` были бы вложены в элементы `Orders`.

Указанная особенность, касающаяся правильного выбора последовательности расположения столбцов в списке выборки, заслуживает особого внимания! В предыдущей главе были приведены примеры кода XML, в которых элементы `Customers` являются родительскими по отношению к элементам `Products`. Было также показано, как оформить те же данные в виде другой иерархии, в которой те же данные входят в состав другой иерархии, — элементы `Products` становятся родительскими по отношению к элементам `Customers`. Чтобы понять, как следует поступать в каждом случае, достаточно подумать о том, какая основная задача должна быть решена с помощью запроса, включающего конструкцию FOR XML. Компоненты списка выборки следует упорядочивать таким образом, чтобы формируемая иерархическая структура документа XML соответствовала его назначению. Безусловно, если полученная иерархическая структура не отвечает вашим требованиям, ее всегда можно преобразовать в другую форму, но почему бы сразу же не обеспечить формирование с помощью СУБД SQL Server именно той структуры, которая требуется!

При использовании опции `AUTO` обнаруживается определенный недостаток, состоящий в том, что в конечном итоге иногда формируется немного более сложная результирующая модель данных XML по сравнению с той, какой она могла бы быть. Кроме того, в настоящее время опция `AUTO` несовместима с конструкцией `GROUP BY`. А преимущество опции `AUTO` заключается в том, что формируемая иерархическая модель более явно соответствует структуре самих данных. Благодаря этому немного упрощается работа программиста в тех ситуациях, когда обнаруживаются более явно выраженные различия между элементами, например, если создается отчет, отсортированный по нескольким признакам (допустим, если элементы `Orders` должны быть представлены в отсортированном виде в элементах `Customers`, которые также отсортированы).

Опция EXPLICIT

Ключевое слово `EXPLICIT`, применяемое для обозначения данной опции, переводится как "явный". Но любопытно то, что это смысловое значение может рассматриваться в основном как приблизительная характеристика тех синтаксических средств, которые могут использоваться в сочетании с этой опцией при формировании запроса. Очевидно, что подготовка запроса с конструкцией `EXPLICIT` требует больше усилий, но наградой за эти усилия становится предоставление большего контроля над тем, что должно быть представлено в виде элемента или атрибута, а также над тем, какие элементы должны быть вложенными по отношению к другим элементам.

Опция `EXPLICIT` позволяет определить каждый уровень иерархии и указать, как должен выглядеть каждый уровень. Для определения иерархии создается структура, которая имеет условное название **универсальной таблицы**. Универсальная таблица во многих отношениях полностью аналогична любому другому результирующему на-

бору, который может быть сформирован в СУБД SQL Server. Обычно такая таблица формируется путем использования операторов UNION для соединения друг с другом данных, полностью представленных в виде одного уровня иерархии, но для выработки окончательного варианта кода XML можно также, например, формировать основной объем данных с помощью пользовательских функций, а затем применять операторы SELECT к этим функциям. Основное различие между универсальной таблицей и результирующим набором (формируемым с помощью обычных запросов) заключается в том, что непосредственно в универсальной таблице должен быть предусмотрен достаточный объем метаданных, для того чтобы СУБД SQL Server могла преобразовать эту универсальную таблицу в документ XML, соответствующий требуемой схеме.

Чтобы было проще понять, что подразумевается под выражением "достаточный объем метаданных", рассмотрим реальную универсальную таблицу, позволяющую убедиться в том, насколько сложными могут оказаться такие метаданные. Эта таблица используется в примере кода, который рассматривается немного позже в данном разделе (табл. 16.1).

Таблица 16.1. Образец универсальной таблицы

Tag	Parent	Customer!1! CustomerID	Customer!1! CompanyName	Order!2! OrderID	Order!2! OrderDate
1	NULL	ALFKI	Alfreds Futterkiste	NULL	NULL
2	1	ALFKI	Alfreds Futterkiste	10643	1997-08-25 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10692	1997-10-03 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10702	1997-10-13 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10835	1998-01-15 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	10952	1998-03-16 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11011	1998-04-09 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11078	1999-05-01 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11079	NULL
2	1	ALFKI	Alfreds Futterkiste	11080	2000-07-22 16:48:00.000
2	1	ALFKI	Alfreds Futterkiste	11081	2000-07-22 00:00:00.000
2	1	ALFKI	Alfreds Futterkiste	11087	2000-08-05 17:37:52.520
1	NULL	ANTON	Antonio Moreno Taqueria	NULL	NULL
2	1	ANTON	Antonio Moreno Taqueria	10365	1996-11-27 00:00:00.000
2	1	ANTON	Antonio Moreno Taqueria	10507	1997-04-15 00:00:00.000

Окончание табл. 16.1

Tag	Parent	Customer!1! CustomerID	Customer!1! CompanyName	Order!2! OrderID	Order!2! OrderDate
2	1	ANTON	Antonio Moreno Taqueria	10535	1997-05-13 00:00:00.000
2	1	ANTON	Antonio Moreno Taqueria	10573	1997-06-19 00:00:00.000
2	1	ANTON	Antonio Moreno Taqueria	10677	1997-09-22 00:00:00.000
2	1	ANTON	Antonio Moreno Taqueria	10682	1997-09-25 00:00:00.000
2	1	ANTON	Antonio Moreno Taqueria	10856	1998-01-28 00:00:00.000

Именно такая универсальная таблица, как показано в табл. 16.1, требуется для того, чтобы с помощью опции EXPLICIT были получены точно такие же результаты, как и при выполнении запроса с опцией AUTO, который рассматривался в последнем примере.

На первый взгляд может показаться, что нет особой необходимости использовать опцию EXPLICIT, если ее применение приводит к получению таких же результатов, как и с помощью AUTO. Но в действительности результаты, полученные с помощью опций EXPLICIT и AUTO, совпадают, в частности, лишь в данном конкретном примере, который был специально подготовлен автором для иллюстрации некоторых функциональных различий нового варианта кода по сравнению с тем вариантом, который рассматривался выше. Но в дальнейшем изложении материала в данном разделе будет показано, что опция EXPLICIT позволяет реализовать дополнительные функции форматирования, неосуществимые с помощью опции AUTO или RAW (но реализуемые с помощью опции PATH), поэтому текущий пример отнюдь не лишен смысла.

Результирующий набор, приведенный в табл. 16.1, имеет несколько описанных ниже особенностей.

- В этом результирующем наборе имеются два специальных столбца с метаданными (**Tag** и **Parent**), которые введены в результирующий набор, поскольку нет другого способа связать метаданные с данными (метаданные невозможно получить из столбцов таблицы).
- Действительные имена столбцов соответствуют специальному формату (который служит также для предоставления дополнительных метаданных).
- Данные упорядочены в соответствии с их иерархией.

Каждый из компонентов метаданных имеет крайне важное значение с точки зрения получения конечных результатов, поэтому, прежде чем приступить к изучению всего примера, рассмотрим, какие данные требуются для формирования окончательного варианта запроса.

Столбцы Tag и Parent

Как было указано в предыдущей главе, элементы могут содержать другие элементы, но не могут перекрываться. Это означает, что документы XML по самому своему характеру подчиняются требованиям, которые естественным образом обеспечивают

создание иерархической структуры (одни элементы полностью содержатся внутри других элементов и благодаря этому по существу создается родительско-дочерняя связь). Столбцы Tag и Parent определяют, где должна находиться каждая строка в иерархии элементов. Каждой строке назначается определенный уровень дескриптора (а этому уровню в дальнейшем присваивается имя элемента); вполне очевидно, что данному уровню соответствует элемент с именем, указанным в столбце Tag. Это означает, что в столбце Parent содержится ссылочная информация, которая указывает, какой уровень является очередным вышестоящим уровнем в иерархии. Благодаря этому СУБД SQL Server получает информацию о том, на каком уровне должно осуществляться вложение данной строки или каким атрибутам должны присваиваться значения полей этой строки (должно ли быть значение представлено в виде элемента или атрибута, можно узнать по имени поля, но об этом речь пойдет в следующем разделе). Если в столбце Parent содержится NULL-значение, то СУБД SQL Server может определить, что данная строка должна стать элементом верхнего уровня или атрибутом этого элемента.

Таким образом, если имеются данные, которые выглядят, как показано в табл. 16.2, то первая строка будет связана с элементом верхнего уровня (станет атрибутом самого внешнего элемента или самим элементом), а вторая строка будет связана с элементом, вложенным в элемент верхнего уровня (поскольку значение Parent второй строки, равное 1, согласуется со значением Tag первой строки).

Таблица 16.2. Универсальная таблица, предназначенная для формирования иерархической структуры

Tag	Parent
1	NULL
2	1

Способы именования столбцов

Откровенно говоря, когда я впервые приступил к изучению конструкции EXPLICIT, эта часть, касающаяся именования столбцов, оказалась для меня наиболее сложной из всех. Безусловно, столбцы Tag и Parent имеют вполне определенное назначение (не зря же этим столбцам присвоены собственные имена), а имена остальных столбцов включают несколько фрагментов метаданных, которые соединены друг с другом в виде одной строки. Единственным обозначением, позволяющим определить, где заканчивается один фрагмент метаданных и начинается другой, служит восклицательный знак (!).

Для именования столбцов применяется примерно такой формат:

```
<element name>!<tag>![<attribute name>][!{element|hide|ID|IDREF|
IDREFS|xml|xmltext|cdata}]
```

В качестве фрагмента <element name>, безусловно, применяется именно то, что под этим подразумевается, — имя элемента в коде XML. Для любого конкретного уровня дескрипторов задано такое условие: после того как определен столбец с некоторым именем, все другие столбцы с тем же дескриптором должны иметь то же имя, что и в предыдущем столбце (столбцах), с тем же номером дескриптора.

Например, если некоторый столбец уже определен как имеющий имя [MyElement!2!MyCol], то другой столбец может быть назван как [MyElement!2!MyOtherCol], но не как [SomeOtherName!2!MyOtherCol].

Дескриптор связывает столбец со строками, имеющими совпадающий номер дескриптора. При обработке в СУБД SQL Server универсальной таблицы вначале считывается номер дескриптора, а затем анализируются столбцы с тем же номером дескриптора. Поэтому при обнаружении в СУБД SQL Server строки, приведенной в табл. 16.3, выполняется проверка номера дескриптора и обнаруживается, что он равен 1. На основании этого СУБД определяет, что должны быть обработаны столбцы с именами Customer!1!CustomerID и Customer!1!CompanyName, однако не следует обрабатывать столбец Order!2!OrderID.

Таблица 16.3. Одна строка универсальной таблицы

Tag	Parent	Customer!1! CustomerID	Customer!1! CompanyName	Order!2! OrderID	Order!2! OrderDate
1	NULL	ALFKI	Alfreds Futterkiste	NULL	NULL

Таким образом, мы подошли к определению понятия имени атрибута, а с этого начинается очередной этап усложнения данного изложения (отметим, что после этого наступит еще один сложный этап описания). Если не задана какая-либо директива (о чем речь пойдет позже), то данный атрибут рассматривается как обязательный и представляет собой имя атрибута XML, для которого данный столбец должен предоставить значение. Атрибут будет представлен в коде XML как часть элемента, имя которого указано в имени столбца.

Если же задана директива, то атрибут может относиться к одной из трех описанных ниже различных категорий.

- Атрибут является запрещенным. Это означает, что место для атрибута должно оставаться пустым (тем не менее для обозначения того места, где должен находиться атрибут, все равно необходимо применить восклицательный знак). Именно такая ситуации возникает при использовании директивы CDATA.
- Атрибут является необязательным. Это означает, что пользователь может задать значение атрибута, но не обязан этого делать. То, что происходит в этом случае, зависит от выбранной директивы.
- Атрибут все еще является обязательным. Данное условие является истинным для директив elements и xml. В этом случае имя атрибута становится именем полностью нового элемента, который создается в результате применения директивы elements или xml.

Изложенные выше сведения о том, как формируются имена столбцов, вполне позволяют приступить к созданию запроса, поэтому рассмотрим на примере, какие результаты могут быть получены при использовании запроса с опцией EXPLICIT.

Прежде всего рассмотрим запрос, позволяющий получить такие же основные данные, как и в примерах запросов с опциями RAW и AUTO. Вполне очевидно, что при использовании опции EXPLICIT код намного усложняется, чем при использовании опций RAW и AUTO. Применение опций RAW и AUTO в основном сводится к тому, что в конце оператора выборки добавляется конструкция FOR XML. А приведенный ниже пример позволяет сразу же убедиться в том, что при использовании опции EXPLICIT приходится полностью пересматривать весь подход к формированию запросов.

Запрос, созданный в соответствии с поставленной задачей, выглядит следующим образом:

```
USE Northwind
SELECT 1                              as Tag,
   NULL                               as Parent,
   Customers.CustomerID               as [Customer!1!CustomerID],
   Customers.CompanyName              as [Customer!1!CompanyName],
      NULL                            as [Order!2!OrderID],
      NULL                            as [Order!2!OrderDate]
FROM Customers
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
UNION ALL
SELECT 2,
     1,
     Customers.CustomerID,
     Customers.CompanyName,
     Orders.OrderID,
     Orders.OrderDate
FROM Customers
JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
ORDER BY [Customer!1!CustomerID], [Order!2!OrderID]
FOR XML EXPLICIT
```

Обратите внимание на то, что конструкция FOR XML применяется только один раз, после последней операции выборки в соединении UNION.

Очевидно, что в данном случае запрос становится гораздо сложнее, чем при использовании других опций, но достаточно лишь внести несколько изменений, и появляется возможность сформировать с помощью запроса с опцией EXPLICIT такие конструкции XML, которые не могут быть получены с помощью опции AUTO.

В качестве довольно простой иллюстрации рассмотрим следующий пример, в котором внесено несколько небольших изменений в требования к формируемому запросу. Допустим, что решено выразить информацию об имени компании, CompanyName, в виде атрибута элемента Order, а не элемента Customer (еще одна возможность состоит в том, чтобы представить эту информацию и так, и иначе). При использовании опции AUTO для решения указанной задачи потребовалось бы применить некоторые сложные конструкции (в ходе обработки каждой строки приходилось бы снова и снова выполнять поиск значения Company с помощью связанного подзапроса, поскольку опция AUTO не позволяет использовать одно и то же значение в нескольких местах). А если применяются дополнительные операции поиска, то код становится очень сложным. В действительности может оказаться, что из-за усложнения кода вообще не удастся решить поставленную задачу. С другой стороны, с помощью опции EXPLICIT все подобные задачи решаются относительно просто (по крайней мере, ненамного сложнее по сравнению со всеми прочими вариантами применения этой опции).

Для решения указанной задачи с применением опции EXPLICIT достаточно лишь снова получить ссылку на столбец CompanyName в списке выборки, но связать новый экземпляр этого объекта с таблицей Orders, а не Customers:

```
USE Northwind
SELECT 1                              as Tag,
  NULL                                as Parent,
  Customers.CustomerID                as [Customer!1!CustomerID],
  Customers.CompanyName               as [Customer!1!CompanyName],
     NULL                             as [Order!2!OrderID],
     NULL                             as [Order!2!OrderDate],
  NULL                                as [Order!2!CompanyName]
FROM Customers
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
UNION ALL
SELECT 2,
     1,
     Customers.CustomerID,
     Customers.CompanyName,
     Orders.OrderID,
     Orders.OrderDate,
  Customers.CompanyName
FROM Customers
JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
ORDER BY [Customer!1!CustomerID], [Order!2!OrderID]
FOR XML EXPLICIT
```

После вызова на выполнение этого запроса формируются в основном такие же результаты, как и прежде, но на этот раз в элементе `Order` формируется искомый дополнительный атрибут:

```
<Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Order OrderID="10643" OrderDate="1997-08-25T00:00:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="10692" OrderDate="1997-10-03T00:00:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="10702" OrderDate="1997-10-13T00:00:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="10835" OrderDate="1998-01-15T00:00:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="10952" OrderDate="1998-03-16T00:00:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="11011" OrderDate="1998-04-09T00:00:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="11078" OrderDate="1999-05-01T00:00:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="11079" CompanyName="Alfreds Futterkiste" />
  <Order OrderID="11080" OrderDate="2000-07-22T16:48:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="11081" OrderDate="2000-07-22T00:00:00"
CompanyName="Alfreds Futterkiste" />
  <Order OrderID="11087" OrderDate="2000-08-05T17:37:52.520"
CompanyName="Alfreds Futterkiste" />
  </Customer>
<Customer CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria">
  <Order OrderID="10365" OrderDate="1996-11-27T00:00:00"
CompanyName="Antonio Moreno Taqueria" />
```

```
  <Order OrderID="10507" OrderDate="1997-04-15T00:00:00"
CompanyName="Antonio Moreno Taqueria" />
  <Order OrderID="10535" OrderDate="1997-05-13T00:00:00"
CompanyName="Antonio Moreno Taqueria" />
  <Order OrderID="10573" OrderDate="1997-06-19T00:00:00"
CompanyName="Antonio Moreno Taqueria" />
  <Order OrderID="10677" OrderDate="1997-09-22T00:00:00"
CompanyName="Antonio Moreno Taqueria" />
  <Order OrderID="10682" OrderDate="1997-09-25T00:00:00"
CompanyName="Antonio Moreno Taqueria" />
  <Order OrderID="10856" OrderDate="1998-01-28T00:00:00"
CompanyName="Antonio Moreno Taqueria" />
  </Customer>
```

Обратите внимание на тот факт, что в некоторых элементах `Order` атрибут `OrderDate` полностью отсутствует. Таковыми являются элементы, соответствующие строкам, в которых значением поля `OrderDate` было `NULL`. В языке XML для представления атрибутов с неопределенными значениями эти атрибуты исключаются.

В действительности данный пример демонстрирует лишь самые основные возможности. А для реализации гораздо более сложных функциональных средств могут использоваться **директивы**, которые позволяют формировать и управлять выводом не только данных, но и информации о схеме (если применяется опция `XMLDATA`).

На первых порах для освоения директив требуются значительные усилия. Но после того как вы научитесь работать с директивами, вскоре обнаружится, что способы их применения являются не такими уж сложными, хотя в отдельных случаях конструкции, в которых используются директивы, все же могут показаться запутанными (действия, осуществляемые с помощью некоторых директив, не очевидны на первый взгляд, кроме того, иногда зависят от определенных дополнительных факторов). По мнению автора (а также представителей одного известного ему коллектива разработчиков, которые думают так же, но никогда в этом не признаются), в действительности корпорацией Microsoft решение о вводе в действие директив было принято несмотря на то, что с их применением связаны значительные трудности, поскольку перспективы организации работы с помощью директив слишком привлекательны, чтобы от них можно было отказаться.

Вообще говоря, предусмотрена возможность использовать с опциями `EXPLICIT` восемь директив. Некоторые из этих директив могут применяться совместно на одном и том же уровне иерархии, а применение других исключает возможность использования определенной части директив на каком-то конкретном уровне иерархии.

Назначение директив состоит в том, что они позволяют модифицировать формат представления результатов. Откровенно говоря, если бы не было возможности вводить директивы в код операторов, то применение опции `EXPLICIT` имело бы мало смысла или вообще было бы бессмысленным. (В действительности опция `AUTO` позволяет выполнить основную часть тех же операций форматирования данных, что и опция `EXPLICIT`, особенно если в сочетании с опцией `EXPLICIT` не используются директивы, даже несмотря на то, что, как уже было сказано выше, иногда для этого приходится применять довольно сложные конструкции с опцией `AUTO`.) Итак, учитывая сделанные здесь замечания, рассмотрим, какие директивы являются доступными.

Директива element

Понять назначение директивы element можно гораздо проще по сравнению со всеми остальными. Назначение этой директивы сводится лишь к тому, что она указывает на необходимость оформить значения в рассматриваемом столбце в виде элементов, а не атрибутов. Содержимое созданного элемента вводится в структуру документа XML в виде дескриптора, дочернего по отношению к текущему дескриптору. Например, предположим, что условия предыдущих примеров были уточнены таким образом, чтобы значение OrderDate было представлено в виде отдельного элемента. Эту задачу можно решить путем добавления директивы element к концу определения поля OrderDate:

```
SELECT 1                          as Tag,
  NULL                            as Parent,
  Customers.CustomerID            as [Customer!1!CustomerID],
  Customers.CompanyName           as [Customer!1!CompanyName],
     NULL                         as [Order!2!OrderID],
     NULL                         as [Order!2!OrderDate!element]
FROM Customers
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
UNION ALL
SELECT 2,
     1,
     Customers.CustomerID,
     Customers.CompanyName,
     Orders.OrderID,
     Orders.OrderDate
FROM Customers
JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
ORDER BY [Customer!1!CustomerID], [Order!2!OrderID]
FOR XML EXPLICIT
```

Этого вполне достаточно, чтобы получить дополнительный элемент вместо атрибута:

```
<Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Order OrderID="10643">
    <OrderDate>1997-08-25T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10692">
    <OrderDate>1997-10-03T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10702">
    <OrderDate>1997-10-13T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10835">
    <OrderDate>1998-01-15T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10952">
    <OrderDate>1998-03-16T00:00:00</OrderDate>
  </Order>
  <Order OrderID="11011">
    <OrderDate>1998-04-09T00:00:00</OrderDate>
```

```
  </Order>
</Customer>
<Customer CustomerID="ANTON" CompanyName="Antonio Moreno Taqueria">
  <Order OrderID="10365">
    <OrderDate>1996-11-27T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10507">
     <OrderDate>1997-04-15T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10535">
    <OrderDate>1997-05-13T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10573">
    <OrderDate>1997-06-19T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10677">
    <OrderDate>1997-09-22T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10682">
    <OrderDate>1997-09-25T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10856">
    <OrderDate>1998-01-28T00:00:00</OrderDate>
  </Order>
</Customer>
```

Директива `xml`

Директива `xml` почти аналогична директиве `element`. Дело в том, что директива `xml` также указывает, что данные столбца, к которому она относится, должны быть сформированы в виде элемента, а не атрибута. Различия между директивами `xml` и `element` обнаруживаются, только если в представляемых данных имеются специальные символы, требующие кодирования. Например, в коде XML знак равенства = является зарезервированным. Если же необходимо представить сам символ =, то следует его **закодировать** (в закодированном виде символ = принимает вид `&eq;`). При использовании директивы `element` кодирование содержимого элемента осуществляется автоматически, а при использовании директивы `xml` содержимое элемента непосредственно передается в результирующий код XML без кодирования. Если применяется директива `xml`, ни одна прочая конструкция на том же уровне (на уровне с тем же номером) не может включать какую-либо директиву, кроме `hide`.

Директива `hide`

Директива `hide` — еще одна простая директива, с помощью которой осуществляется именно то, что следует из ее названия, — она скрывает результаты, представленные в конкретном столбце.

Необходимость в таком сокрытии данных возникает в связи с тем, что иногда приходится включать в запрос столбцы в целях выполнения с данными этих столбцов каких-то других операций, отличных от операции вывода. Если в качестве примера взять обычный запрос, то, как известно, в запросе иногда приходится выполнять сортировку с помощью конструкции `ORDER BY` с учетом данных таких столбцов, которые не присутствуют в списке выборки. Но при выполнении запросов с операциями `UNION` возможность указывать только нужные столбцы отсутствует, поскольку в

операции объединения могут участвовать лишь те столбцы, которые заданы в списке выборки.

Рассмотрим небольшой пример, в котором отслеживаются некоторые результаты сбыта товаров, названия которых даны в столбце `Product`. Предположим, что требуется получить список всех товаров, определить идентификаторы `OrderID` тех заказов, согласно которым была выполнена их поставка, а также узнать, в какие даты осуществлялись эти поставки. Требуется получить только идентификаторы товаров `ProductID`, но значения `ProductID` должны быть отсортированы так, чтобы данные обо всех конкретных товарах находились рядом с данными об аналогичных товарах. Это означает, что должна быть выполнена сортировка по идентификаторам категории `CategoryID`, но сами значения `CategoryID` не должны быть включены в окончательные результаты.

Вначале сформируем запрос без директивы `hide`; это позволяет проверить, как осуществляется намеченная сортировка:

```
SELECT 1                              as Tag,
  NULL                                as Parent,
  ProductID                           as [Product!1!ProductID],
    CategoryID                        as [Product!1!CategoryID],
  NULL                                as [Order!2!OrderID],
  NULL                                as [Order!2!OrderDate]
FROM Products
UNION ALL
SELECT 2,
  1,
  p.ProductID,
     p.CategoryID,
  od.OrderID,
  o.OrderDate
FROM Products AS p
JOIN  [Order Details] AS od
 ON p.ProductID = od.ProductID
JOIN Orders AS o
 ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
ORDER BY [Product!1!CategoryID],[Product!1!ProductID], [Order!2!OrderID]
FOR XML EXPLICIT
```

Обязательно ознакомьтесь с тем, как осуществляется обработка значения `OrderDate` в данном запросе. Безусловно, требовалось осуществить выборку этой информации из таблицы `Orders`, но данную информацию можно было легко объединить со значениями идентификаторов `OrderID`, полученных из таблицы `Order Details` (хотя бы даже потому, что применяется опция `EXPLICIT`). Как оказалось, можно было бы также получить значения идентификаторов `OrderID` из таблицы `Orders`, но иногда приходится смешивать данные из нескольких таблиц в одном элементе, и настоящий запрос может служить еще одной демонстрацией того, как решить именно эту задачу.

Приведенные ниже результаты показывают, что была выполнена именно такая сортировка, на осуществление которой мы рассчитывали.

```
<Product ProductID="1" CategoryID="1" />
<Product ProductID="2" CategoryID="1">
<Order OrderID="10813" OrderDate="1998-01-05T00:00:00" />
</Product>
```

```
<Product ProductID="24" CategoryID="1" />
<Product ProductID="34" CategoryID="1" />
<Product ProductID="35" CategoryID="1" />
<Product ProductID="38" CategoryID="1">
<Order OrderID="10816" OrderDate="1998-01-06T00:00:00" />
<Order OrderID="10817" OrderDate="1998-01-06T00:00:00" />
</Product>
<Product ProductID="39" CategoryID="1" />
<Product ProductID="43" CategoryID="1">
<Order OrderID="10814" OrderDate="1998-01-05T00:00:00" />
<Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="67" CategoryID="1" />
<Product ProductID="70" CategoryID="1">
<Order OrderID="10810" OrderDate="1998-01-01T00:00:00" />
</Product>
<Product ProductID="75" CategoryID="1">
<Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="76" CategoryID="1">
<Order OrderID="10808" OrderDate="1998-01-01T00:00:00" />
</Product>
<Product ProductID="3" CategoryID="2" />
<Product ProductID="4" CategoryID="2" />
...
```

Обратите внимание на то, что приведенные результаты были немного сокращены в целях экономии места.

Теперь введем директиву `hide`, чтобы исключить информацию о категории:

```
SELECT 1                          as Tag,
  NULL                            as Parent,
  ProductID                       as [Product!1!ProductID],
     CategoryID                   as [Product!1!CategoryID!hide],
  NULL                            as [Order!2!OrderID],
  NULL                            as [Order!2!OrderDate]
FROM Products
UNION ALL
SELECT 2,
  1,
  p.ProductID,
     p.CategoryID,
  od.OrderID,
  o.OrderDate
FROM Products AS p
JOIN  [Order Details] AS od
 ON p.ProductID = od.ProductID
JOIN Orders AS o
 ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
ORDER BY [Product!1!CategoryID!hide],[Product!1!ProductID],
[Order!2!OrderID]
FOR XML EXPLICIT
```

Очевидно, что формируются такие же результаты, но на этот раз информация столбца Category действительно скрыта:

```
<Product ProductID="1" />
<Product ProductID="2">
<Order OrderID="10813" OrderDate="1998-01-05T00:00:00" />
</Product>
<Product ProductID="24" />
<Product ProductID="34" />
<Product ProductID="35" />
<Product ProductID="38">
<Order OrderID="10816" OrderDate="1998-01-06T00:00:00" />
<Order OrderID="10817" OrderDate="1998-01-06T00:00:00" />
</Product>
<Product ProductID="39" />
<Product ProductID="43">
<Order OrderID="10814" OrderDate="1998-01-05T00:00:00" />
<Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="67" />
<Product ProductID="70">
<Order OrderID="10810" OrderDate="1998-01-01T00:00:00" />
</Product>
<Product ProductID="75">
<Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="76">
<Order OrderID="10808" OrderDate="1998-01-01T00:00:00" />
</Product>
<Product ProductID="3" />
<Product ProductID="4" />
...
```

Директивы id, idref и idrefs

Ни одна из директив id, idref и idrefs не выполняет каких-либо действий, если в сочетании с ними не используется также опция XMLDATA (которая должна следовать за опцией EXPLICIT в конструкции FOR XML). Еще одна особенность применения этих директив состоит в том, что с их помощью формируются документы, предназначенные для проверки по схеме, которая содержит соответствующие объявления. Причина, по которой применяется такая организация работы, становится вполне очевидной после анализа назначения этих директив. Ведь с их помощью вводятся дополнения к схеме, позволяющие провести проверку определенных характеристик документа, но само проведение такой проверки без схемы является невозможным.

Дело в том, что применение конструкции FOR XML обеспечивает возможность включения в документ атрибутов id. Такие атрибуты id выполняют в документе XML функции, аналогичные функциям первичных ключей в реляционных данных, поскольку с их помощью назначается уникальный идентификатор для элемента документа XML, с именем которого связан идентификатор id. Но ни для одного имени элемента не допускается указание в определении id больше одного атрибута. Имя атрибута, предназначенного для использования в качестве идентификатора id, определяется в схеме для формируемого документа XML. А после того как будет объявлен один элемент с указанным значением в качестве атрибута id, становится невозмож-

ным создание какого-либо элемента с тем же именем, имеющего такое же значение атрибута.

> Следует отметить, что, в отличие от первичных ключей, применяемых в реляционных объектах и обрабатываемых с помощью операторов SQL, в документе XML не допускается применение нескольких атрибутов в составе идентификатора `id`.

Таким образом, конструкция `FOR XML` позволяет формировать атрибуты, которые по своему назначению аналогичны первичным ключам, поэтому не удивительно, что конструкция `FOR XML` позволяет также формировать атрибуты, которые могут применяться по аналогии с внешними ключами. Для этой цели предназначены директивы `idref` и `idrefs`. Обе эти директивы служат для создания ссылки, связывающей определенный атрибут одного элемента с атрибутом `id` другого элемента.

С точки зрения формирования структуры документа это означает, что с помощью директив `id`, `idref` и `idrefs` могут быть созданы отношения между элементами, в частности отношения вложения. После того как атрибуту `id` одного элемента присваивается определенное значение, а затем формируется ссылка на этот идентификатор из другого элемента с помощью атрибута, объявленного как принадлежащий к категории атрибутов `idref` или `idrefs`, появляется возможность определения связи между этими двумя элементами, независимо от их позиции в документе.

Необходимость в использовании для формирования подобных связей двух директив, `idref` и `idrefs`, обусловлена тем, что формируемые связи могут относиться к разным типам. Атрибут `idref` может содержать только одно значение, которое должно согласовываться со значением атрибута `id` существующего элемента, а атрибут `idrefs` может содержать многозначное значение, представляющее собой список значений, разделенных пробельными символами, причем и эти значения должны быть таковыми, что каждое из них согласуется со значением атрибута `id` одного из существующих элементов. Таким образом, атрибуты `idref` используются, если есть необходимость установить связь "один ко многим" (возможно наличие в документе только по одному из всех значений `id`, т.е. значения `id` должны быть уникальными, но количество элементов с тем же значением атрибута `idref` не ограничивается). С другой стороны, атрибуты `idrefs` используются, если необходимо установить связь "многие ко многим" (каждый элемент с атрибутом `idrefs` может содержать ссылки на многие элементы с атрибутами `id`, указанными в значении `idrefs`, а само количество элементов с атрибутами `idrefs`, в которых указаны одни и те же значения `id`, не ограничено).

Чтобы проиллюстрировать использование рассматриваемых директив, необходимо внести небольшие изменения в предыдущий запрос. Вначале введем в действие директиву `idref`:

```
SELECT 1                          as Tag,
  NULL                                 as Parent,
  ProductID                             as [Product!1!ProductID!ID],
    CategoryID                          as [Product!1!CategoryID!hide],
  NULL                            as [Order!2!OrderID],
  NULL                            as [Order!2!ProductID!idref],
  NULL                            as [Order!2!OrderDate]
FROM Products
UNION ALL
```

```
SELECT 2,
  1,
  p.ProductID,
     p.CategoryID,
  od.OrderID,
  od.ProductID,
  o.OrderDate
FROM Products AS p
JOIN  [Order Details] AS od
 ON p.ProductID = od.ProductID
JOIN Orders AS o
 ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
ORDER BY [Product!1!CategoryID!hide],[Product!1!ProductID!ID],
[Order!2!OrderID]
FOR XML EXPLICIT, XMLDATA
```

После просмотра полученных результатов обнаруживается, что фактически в них интерес представляют только два фрагмента — схема и элемент `Product`:

```
<Schema name="Schema2" xmlns="urn:schemas-microsoft-com:xml-data" xmlns:
dt="urn:schemas-microsoft-com:datatypes">
<ElementType name="Product" content="mixed" model="open">
<AttributeType name="ProductID" dt:type="id" />
<attribute type="ProductID" />
</ElementType>
<ElementType name="Order" content="mixed" model="open">
<AttributeType name="OrderID" dt:type="i4" />
<AttributeType name="ProductID" dt:type="idref" />
<AttributeType name="OrderDate" dt:type="dateTime" />
<attribute type="OrderID" />
<attribute type="ProductID" />
<attribute type="OrderDate" />
</ElementType>
</Schema>
```

В самой схеме присутствует некоторая довольно характерная информация о типе. Элемент `Product` объявлен как принадлежащий к типу элемента. Кроме того, обнаруживается, что элемент `ProductID` объявлен как выполняющий функции атрибута `id` для элемента этого типа. Аналогичным образом, элемент `Order` объявлен как имеющий атрибут `ProductID`, который относится к типу `idref`.

Следующим фрагментом, представляющим интерес, является элемент `Product`:

```
<Product xmlns="x-schema:#Schema2" ProductID="2">
<Order OrderID="10813" ProductID="2" OrderDate="1998-01-05T00:00:00"/>
</Product>
```

В данном случае заслуживает внимания то, что в коде, сформированном в СУБД SQL Server, имеется ссылка на вложенную схему в объявлении элемента `Product`. Это равносильно объявлению о том, что элемент `Product` и все, что к нему относится, должно соответствовать схеме; таким образом, дается гарантия, что значения атрибутов `id` и `idref` всегда будут согласованы.

При осуществлении попытки применения директивы `idrefs` задача немного усложняется. Дело в том, что в СУБД SQL Server для формирования списка значений атрибутов `idrefs` требуется применение запроса, отдельного от того запроса,

с помощью которого формируются элементы с атрибутами id. А в рассматриваемом примере мы должны выполнить не только это требование, но и ввести еще один запрос в состав конструкции с операцией UNION для подготовки значений атрибутов idrefs (поскольку список возможных значений атрибутов id должен быть известен до того, как мы приступим к формированию списка значений атрибутов idrefs, но фактические значения атрибутов id станут известны только после полного формирования атрибутов id). Запрос на формирование значений атрибутов idrefs должен непосредственно предшествовать запросу, в котором формируются значения атрибутов id. Таким образом, в своей окончательной форме запрос становится довольно замысловатым:

```
SELECT 1                          as Tag,
  NULL                            as Parent,
  ProductID                       as [Product!1!ProductID],
     CategoryID                   as [Product!1!CategoryID!hide],
  NULL                            as [Product!1!OrderList!idrefs],
  NULL                            as [Order!2!OrderID!id],
  NULL                            as [Order!2!OrderDate]
FROM Products
UNION ALL
SELECT 1,
   NULL,
   p.ProductID,
      p.CategoryID,
   'id' + CAST(od.OrderID AS varchar),
   NULL,
   NULL
FROM Products AS p
JOIN  [Order Details] AS od
 ON p.ProductID = od.ProductID
JOIN Orders AS o
 ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
UNION ALL
SELECT 2,
   1,
   p.ProductID,
      p.CategoryID,
   NULL,
   'id' + CAST(od.OrderID AS varchar),
   o.OrderDate
FROM Products AS p
JOIN  [Order Details] AS od
 ON p.ProductID = od.ProductID
JOIN Orders AS o
 ON od.OrderID = o.OrderID
WHERE o.OrderDate BETWEEN '1998-01-01' AND '1998-01-07'
ORDER BY [Product!1!CategoryID!hide],[Order!2!OrderID!id], [Product!1!Orde
rList!idrefs]
FOR XML EXPLICIT, XMLDATA
```

Но полученная схема в конечном итоге во многом напоминает схему, которая была получена при использовании директивы idref:

```
<Schema name="Schema9" xmlns="urn:schemas-microsoft-com:xml-data" xmlns:
dt="urn:schemas-microsoft-com:datatypes">
<ElementType name="Product" content="mixed" model="open">
<AttributeType name="ProductID" dt:type="i4" />
<AttributeType name="OrderList" dt:type="idrefs" />
<attribute type="ProductID" />
<attribute type="OrderList" />
</ElementType>
<ElementType name="Order" content="mixed" model="open">
<AttributeType name="OrderID" dt:type="id" />
<AttributeType name="OrderDate" dt:type="dateTime" />
<attribute type="OrderID" />
<attribute type="OrderDate" />
</ElementType>
</Schema>
```

Тем не менее различия между элементами становятся весьма существенными:

```
<Product xmlns="x-schema:#Schema9" ProductID="76" OrderList="id10808
id10810 id10813 id10814 id10816 id10817 id10819 id10819">
  <Order OrderID="id10808" OrderDate="1998-01-01T00:00:00"/>
  <Order OrderID="id10810" OrderDate="1998-01-01T00:00:00"/>
  <Order OrderID="id10813" OrderDate="1998-01-05T00:00:00"/>
  <Order OrderID="id10814" OrderDate="1998-01-05T00:00:00"/>
  <Order OrderID="id10816" OrderDate="1998-01-06T00:00:00"/>
  <Order OrderID="id10817" OrderDate="1998-01-06T00:00:00"/>
  <Order OrderID="id10819" OrderDate="1998-01-07T00:00:00"/>
  <Order OrderID="id10819" OrderDate="1998-01-07T00:00:00"/>
</Product>
```

Таким образом, при использовании директив `id`, `idref` и `idrefs` приходится решать очень сложные задачи. Но благодаря этому появляется возможность обеспечить строгий контроль типов выходных данных. Разумеется, в большинстве практических ситуаций просто нет необходимости добиваться столь высокого уровня контроля и брать на себя все связанные с этим сложности, но бывают и такие обстоятельства, в которых без применения этих трех директив буквально невозможно обойтись.

Директива `xmltext`

Директива `xmltext` применяется с учетом того, что содержимое столбца, к которому она относится, представляет собой код XML. В результате выполнения этой директивы предпринимается попытка вставить такой код XML как неотъемлемую часть создаваемого документа XML.

На первый взгляд может показаться, что действия этой директивы осуществляются достаточно просто, ведь с ее помощью в документ должен лишь быть вставлен некоторый текст, но фактически предусмотрены описанные ниже достаточно строгие правила, которые определяют, где, когда и как должна происходить вставка данных.

- При условии, что код XML, попытка вставки которого предпринимается с помощью директивы `xmltext`, является формально правильным, корневой элемент должен быть удален, а атрибуты этого элемента — сохранены и применены с учетом нескольких последующих правил.
- Если при использовании директивы `xmltext` не задано имя атрибута, то сохранившиеся атрибуты удаленного корневого элемента должны быть введены в состав элемента, содержащего директиву `xmltext`. Сохранившиеся атрибуты

будут использоваться в скомбинированном таким образом элементе со своими именами. Если имена каких-либо атрибутов, сохранившихся после удаления корневого элемента, конфликтуют с другой информацией атрибутов в комбинированном элементе, то конфликтующие атрибуты удаляются из сохранившихся данных.

- Все элементы, вложенные в удаляемый элемент, становятся вложенными элементами комбинированного элемента.
- Если в директиве `xmldata` указано имя атрибута, то сохранившиеся данные помещаются в элемент с указанным именем. Новый элемент становится дочерним по отношению к элементу, в котором содержится директива.
- Если какая-либо часть полученного в результате кода XML не является формально правильной, то поведение средств обработки директивы `xmldata` не определено. По существу, это поведение зависит от того, как будут выглядеть конечные результаты, но, скорее всего, произойдет формирование сообщения об ошибке. (Автору еще не приходилось сталкиваться с такой ситуацией, в которой бы применялась ссылка на данные, не являющиеся формально правильными, и при этом удавалось избежать возникновения ошибки.)

Директива cdata

Сам принцип включения в документ данных с помощью директивы `cdata` впервые был применен в определениях DTD и в языке SGML. По существу, директива `cdata` (сокращение от character data) предназначена для включения в документ символьных данных. В синтаксических анализаторах XML данные, содержащиеся в разделах `cdata`, полностью игнорируются. Проверка данных, находящихся в разделе `cdata`, не выполняется, поэтому нет необходимости применять для данных какую-либо кодировку, соответствующую требованиям XML, т.е. данные могут быть совершенно произвольными. Директива `cdata` используется во всех тех случаях, когда данные, обрабатываемые в составе документа XML, должны остаться полностью неизменными (в частности, не требуется переводить данные в другую кодировку). Еще одним примером применения этой директивы может служить такая ситуация, когда требуется с помощью средств XML передать данные из одного места в другое, но невозможно заранее определить, каковыми являются эти данные (поэтому предугадать, будут ли возникать проблемы при их обработке и передаче или все пройдет успешно). Еще один, далеко не последний пример, может состоять в том, что иногда требуется передать с помощью документа XML двоичную информацию, закодировав ее в символьном виде (с помощью MIME, Base64 или какой-то другой схемы кодировки), после чего ввести полученные результаты в секцию `CDATA`.

В качестве иллюстрации применения директивы рассмотрим достаточно простой пример — таблицу `Employees` базы данных `Northwind`. В этой таблице имеется столбец, относящийся к типу данных `text`. По существу, содержимое этого столбца ничем не регламентировано, поскольку в нем содержатся примечания к данным о служащих. Для преобразования текста, содержащегося в этих примечаниях, в коде XML можно применить примерно такой запрос:

```
SELECT 1                    as Tag,
     NULL                   as Parent,
     Employees.EmployeeID   as [Employee!1!EmployeeID],
     Employees.Notes        as [Employee!1!!cdata]
```

```
FROM Employees
ORDER BY [Employee!1!EmployeeID]
FOR XML EXPLICIT
```

Выходные данные этого запроса являются довольно несложными:

```
<Employee EmployeeID="1">
<![CDATA[
Education includes a BA in psychology from Colorado State University in
1970.  She also completed "The Art of the Cold Call."  Nancy is a member of
Toastmasters International.
]]>
</Employee>
<Employee EmployeeID="2">
<![CDATA[
Andrew received his BTS commercial in 1974 and a Ph.D. in international
marketing from the University of Dallas in 1981.  He is fluent in French and
Italian and reads German.  He joined the company as a sales representative,
was promoted to sales manager in January 1992 and to vice president of sales
in March 1993.  Andrew is a member of the Sales Management Roundtable, the
Seattle Chamber of Commerce, and the Pacific Rim Importers Association.
]]>
</Employee>
<Employee EmployeeID="3">
<![CDATA[
Janet has a BS degree in chemistry from Boston College (1984).  She has also
completed a certificate program in food retailing management.  Janet was
hired as a sales associate in 1991 and promoted to sales representative in
February 1992.
]]>
</Employee>
```

Анализ элемента с атрибутом `EmployeeID="1"` позволяет сразу же обнаружить, что в нем для представления данных не используется какая-либо кодировка (поскольку в противном случае символы кавычек (") были бы представлены в виде `&quot;`).

Итак, очевидно, что применение директивы `cdata` является довольно простым.

Опция PATH

В предыдущих разделах рассматривались конструкции, которые в большей степени соответствуют подходу, применяемому в программном обеспечении корпорации Microsoft, а данный раздел посвящен описанию способов обработки данных, главным образом соответствующих подходу, предусмотренному в самом языке XML. Этот подход основан на использовании опции `PATH`.

Опция `PATH` пришла на смену опции `EXPLICIT`. Безусловно, опция `EXPLICIT` еще не рассматривается как устаревшая, но на это не следует полагаться, поскольку опция `PATH` фактически предназначена для осуществления более эффективных методов решения тех задач, которые в свое время могли быть решены только с помощью опции `EXPLICIT`. Но опция `PATH` имеет более широкую область применения, поэтому в большинстве случаев рекомендуется использовать для формирования документов XML со сложной структурой именно ее.

Но фактически задача выполнения этой рекомендации сложнее, чем может показаться на первый взгляд, несмотря на то, что в документации, подготовленной корпорацией Microsoft, утверждается, что по своему применению опция PATH проще, чем EXPLICIT. Безусловно, опция PATH позволяет создавать программные конструкции, более простые в некоторых отношениях, но, как будет показано в данном разделе, при ее применении приходится сталкиваться с такими многочисленными исключениями, что в конечном итоге становится нелегко понять, каким должен быть окончательный вариант запроса. Дело в том, что для успешного использования опции PATH необходимо знать язык XPath, а если разработчик не обладает такими знаниями, то для него фактически во многих случаях более простой в применении становится опция EXPLICIT. Тем не менее специалисты, использующие в своей работе язык XML, так или иначе обязаны освоить язык XPath, а после успешного его освоения обнаруживается, что подход, основанный на применении опции PATH, который базируется на знании языка XPath, является более удобным.

Однако следует учитывать, что если в разрабатываемом приложении должна обеспечиваться обратная совместимость с программным обеспечением версии SQL Server 2000, то допускается применение только опции EXPLICIT.

Несмотря на вышесказанное, в наиболее простых вариантах ее использования опция PATH является не такой уж сложной. Поэтому вначале рассмотрим, какие основные требования связаны с применением опции PATH. Затем перейдем к рассмотрению более сложных примеров, которые позволяют продемонстрировать некоторые нетривиальные возможности опции PATH.

Общее описание условий использования опции PATH

Опция PATH основана на модели, результаты применения которой должны соответствовать существующему стандарту получения данных из документа XML — стандарту XPath. Прежде всего следует отметить, что XPath — это язык, сформулированный в общепринятом стандарте, который предоставляет способ обозначения путей к конкретным компонентам древовидной структуры документа XML. В том подходе к обработке данных, который основан на использовании опции PATH, применяются в основном те же правила и понятия, определяющие способы выборки данных из документа XML, которые уже рассматривались в настоящей главе.

Способы формирования ссылок на конкретные данные с помощью опции PATH регламентируются несколькими правилами, в которых, в частности, учитывается то, является ли столбец с данными именованным или неименованным (как и при использовании опции EXPLICIT, если задан псевдоним, то именем считается этот псевдоним). Если столбец имеет имя, то применяется еще несколько дополнительных правил в зависимости от обстоятельств.

В следующих разделах рассматриваются некоторые варианты применения опции PATH.

Описание языка XPath представляет собой отдельную тему, которой посвящены целые книги. В опции PATH используется широкий набор средств, доступных в языке XPath, поэтому для описания всех возможностей этой опции недостаточно посвятить лишь одну главу в учебнике для начинающих. Тем не менее в этом разделе приведены необходимые основы, а в следующем разделе читателю предоставляется возможность ознакомиться с более сложным материалом. Фактически этого должно быть достаточно, чтобы приступить к более полному изучению языка XPath, а также понять, какие области применения опции PATH заслуживают большего внимания. Кроме того, отметим, что более полное описание двух указанных тем будет приведено в следующей книге серии для профессионалов — Professional SQL Server 2005 Programming.

Неименованные столбцы

Данные, находящиеся в тех столбцах результирующего набора, которым не присвоены имена, рассматриваются в элементе `row` как бесформатный текст. В качестве примера использования неименованных столбцов рассмотрим модифицированную версию запроса, который применялся для иллюстрации конструкции `FOR XML RAW`. Предположим, что в данном примере необходимо получить список, содержащий сведения о двух интересующих нас заказчиках и номерах размещенных ими заказов:

```
USE Northwind
SELECT Customers.CustomerID,
  COUNT(Orders.OrderID)
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

Рассмотрим полученные результаты:

```
<row><CustomerID>ALFKI</CustomerID>7</row>
<row><CustomerID>ANTON</CustomerID>7</row>
```

В данном случае был создан элемент `row` для каждой строки, полученной в результате запроса (во многом аналогично тому, как и при использовании опции `RAW`), но здесь заслуживает внимания то, что обработка данных одного из столбцов выполнена по-другому.

Очевидно, что столбец `CustomerID` имеет имя, поэтому данные этого столбца помещены в отдельный, собственный элемент (дополнительная информация об этом будет приведена в следующем разделе), но в полученных результатах следует также обратить внимание на число `7`. Эта цифра `7` представляет собой всего лишь произвольно вставленный текст, определяющий содержимое элемента `row`; этот текст даже не ассоциируется непосредственно с элементом `CustomerID`, поскольку лежит за пределами элемента `CustomerID`.

Мне неловко повторять одно и то же бесконечное количество раз, но я обязан снова напомнить, что точные числовые значения (в данном случае два числа `7`), полученные в конкретной системе, зависят от того, много ли было проведено экспериментов с данными таблицы `Orders`. В этом случае важно не то, какие именно числовые данные формируются в сочетании с элементом `CustomerID`, а то, что в элементе `raw` создается бесформатный текст.

Но практика показывает, что ситуации, в которых целесообразно применять произвольный текст на уровне верхнего элемента, встречаются довольно редко. Безусловно, никто не утверждает, что это недопустимо, но, по-видимому, назначение данных, не обозначенных дескриптором, не совсем очевидно. Тем не менее указанный пример приведен исключительно в качестве иллюстрации, а возможность применения подобного подхода зависит от потребностей конкретной системы.

Именованные столбцы

Усложнение используемых программных конструкций происходит значительно быстрее именно после перехода к применению именованных столбцов. В своей наиболее простой форме именованные столбцы применяются почти также просто,

как неименованные. И действительно, один из вариантов использования именованного столбца был показан в предыдущем примере. Дело в том, что если некоторый столбец, имеющий имя, просто обозначается с помощью опции PATH, то происходит всего лишь добавление в элемент row дополнительного элемента, соответствующего этому столбцу:

```
<row><CustomerID>ALFKI</CustomerID>7</row>
```

В данном случае столбец CustomerID представляет собой простой именованный столбец.

Но предусмотрена также возможность ввести в обозначение имени столбца дополнительные символы для указания на то, что для этого столбца требуется специальная обработка. Некоторые из наиболее важных символов рассматриваются в следующих разделах.

Символ @

Данный раздел посвящен описанию символа @. Если к имени столбца добавляется символ @, то этот столбец рассматривается в СУБД SQL Server как атрибут предыдущего столбца. Преобразуем структуру формируемого фрагмента XML так, чтобы данные столбца CustomerID были представлены в виде атрибута элемента верхнего уровня, содержащего данные строки:

```
SELECT Customers.CustomerID AS '@CustomerID',
  COUNT(Orders.OrderID)
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

В результате формируется следующий вывод:

```
<row CustomerID="ALFKI">7</row>
<row CustomerID="ANTON">7</row>
```

Обратите внимание на то, что данные о количестве заказов по-прежнему представлены в виде текстового информационного наполнения элемента row, а в состав самого элемента вошли в виде атрибута только данные указанного столбца. Такое преобразование можно продолжить, присвоив имя столбцу с данными о количестве заказов и также обозначив его имя префиксом для преобразования в атрибут:

```
SELECT Customers.CustomerID AS '@CustomerID',
  COUNT(Orders.OrderID) AS '@OrderCount'
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

Таким образом, в элементе больше не остается произвольного текста:

```
<row CustomerID="ALFKI" OrderCount="7"/>
<row CustomerID="ANTON" OrderCount="7"/>
```

Следует также отметить, что в СУБД SQL Server предусмотрена возможность учесть то, что все содержимое элемента перешло в атрибуты, и в этом элементе больше нет ни элементов более низкого уровня, ни даже простого текста, поэтому вместо элемента с открывающим и закрывающим дескриптором используется элемент, содержащий признак закрытия в самом себе (обратите внимание на знак / в конце элемента).

Очевидно, что все программные конструкции, применявшиеся до сих пор, были довольно простыми. Тем не менее возможность применения этих конструкций зависит от того, выполняются ли некоторые важные правила. Чтобы показать, в чем состоят эти правила, введем небольшое изменение в первоначальный запрос. На первый взгляд такие изменения являются вполне допустимыми, но при попытке вызвать этот запрос на выполнение СУБД SQL Server вырабатывает сообщение об ошибке:

```
SELECT Customers.CustomerID,
  COUNT(Orders.OrderID) AS '@OrderCount'
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

В данном случае было лишь решено снова предусмотреть оформление значений `CustomerID` в виде отдельного элемента. На первый взгляд может показаться, что благодаря этому должен быть сформирован элемент `CustomerID`, в котором данные столбца `OrderCount` оформляются в виде атрибута, но эта попытка не совсем удается:

```
Msg 6852, Level 16, State 1, Line 3
Attribute-centric column '@OrderCount' must not come after a non-attribute-
centric sibling in XML hierarchy in FOR XML PATH.
```

Анализ возникшей при этом ситуации показывает, что СУБД фактически не может определить, атрибутом какого элемента должны стать данные столбца `OrderCount`, — атрибутом элемента `row` или `CustomerID`.

Символ косой черты

Символ / (косая черта), во многом аналогично `@`, представляет собой специальный символ, который указывает, какие особые операции обработки документа должны быть выполнены. По существу, этот символ используется для определения своего рода пути доступа через иерархию документа, связывающего указанный элемент с теми компонентами документа, которые должны быть ассоциированы с ним. Этот символ может находиться в имени столбца на любой позиции, кроме первой. Для демонстрации применения символа / воспользуемся последним примером (который окончился неудачей) и введем его в том месте, где нужно выполнить поиск информации, при попытке получения которой возникла ошибка.

Прежде всего необходимо внести изменения в конструкцию доступа к столбцу `OrderID`, чтобы получить информацию о том, какому элементу принадлежит этот столбец:

```
SELECT Customers.CustomerID,
  COUNT(Orders.OrderID) AS 'CustomerID/OrderCount'
FROM Customers
```

```
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

Строка, в которой находится символ /, а перед этим символом введено имя поля CustomerID, служит для СУБД SQL Server указанием на то, что поле OrderCount находится в иерархии ниже поля CustomerID. Поскольку предусмотрено много разных способов структуризации иерархии XML, рассмотрим, какие действия выполняются в СУБД SQL Server при осуществлении именно этого способа:

```
<row><CustomerID>ALFKI<OrderCount>7</OrderCount></CustomerID>
</row><row><CustomerID>ANTON<OrderCount>7</OrderCount></CustomerID></row>
```

Напомним, что в предыдущем примере было решено ввести значение OrderCount в качестве атрибута элемента CustomerID, поэтому полученные результаты не соответствуют требованиям, которые были поставлены в этой задаче. Чтобы достичь намеченной цели, можно применить комбинацию символов / и @, но для этого необходимо определить всю иерархию. Однако, поскольку вполне можно предположить, что эта задача окажется сложной, разделим ее решение на два этапа, но вначале проверим такой вариант решения, который кажется подходящим для данного случая (однако приводит к получению фактически такого же сообщения об ошибке, как и в предыдущем примере):

```
SELECT Customers.CustomerID,
  COUNT(Orders.OrderID) AS 'CustomerID/@OrderCount'
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

На этот раз сообщение об ошибке выглядит таким образом:

```
Msg 6852, Level 16, State 1, Line 1
Attribute-centric column 'CustomerID/@OrderCount' must not come after a non-
attribute-centric sibling in XML hierarchy in FOR XML PATH.
```

Прежде чем приступить к устранению этой ошибки, рассмотрим внимательно, какие действия осуществляются при формировании дескрипторов XML с помощью конструкции FOR XML. Ключом к изучению выполняемых при этом действий становится понимание того, что дескрипторы по существу формируются в порядке их указания. Поэтому если требуется ввести в элемент атрибуты, то необходимо учесть, что атрибуты входят в состав дескриптора элемента, а это означает, что все атрибуты должны быть определены до того, как будет задано все прочее содержимое элемента (субэлементы или бесформатный текст).

В данном случае решено помещать информацию поля CustomerID в виде бесформатного текста, но значение OrderCount должно быть представлено в виде атрибута (следует отметить, что в реальной практике данные операции могут выполняться в обратном порядке, но об этом немного позже). Это означает, что в СУБД SQL Server необходимые сведения передаются в обратном порядке. К тому времени как обнару-

живается информация OrderCount, обработка атрибутов элемента CustomerID уже заканчивается и возврат назад становится невозможным.

Поэтому, чтобы исправить текущую ошибку, достаточно передать СУБД SQL Server информацию об атрибутах, прежде чем привести сведения о каких-либо других элементах или бесформатном тексте:

```
SELECT COUNT(Orders.OrderID) AS 'CustomerID/@OrderCount',
  Customers.CustomerID AS 'CustomerID'
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
GROUP BY Customers.CustomerID
FOR XML PATH
```

Такая конструкция применяемых операторов может показаться противоречащей здравому смыслу, но и в этом случае достаточно представить себе, в каком порядке происходит формирование конструкций в коде XML. Вначале формируются атрибуты, и лишь тогда и только тогда мы можем перейти к оформлению информации низкого уровня в элементе CustomerID. Вызвав на выполнение приведенный выше вариант оператора, можно убедиться в том, что он позволяет достичь намеченной цели:

```
<row><CustomerID OrderCount="7">ALFKI</CustomerID></row>
<row><CustomerID OrderCount="7">ANTON</CustomerID></row>
```

Теперь информация о количестве заказов переместилась в позицию атрибута OrderCount, как и было задумано, а фактические данные об идентификаторе заказчика CustomerID по-прежнему представлены в виде бесформатного текста, вложенного в элемент.

Прослеживайте до конца логику упорядочения компонентов структуры, которая должна быть сформирована, поскольку этот подход позволяет успешно решать практически любые задачи. В частности, если бы потребовалось также задать значение CustomerID в виде атрибута, а не бесформатного текста, но так, чтобы этот атрибут находился после атрибута OrderCount, то и данную задачу можно было бы легко решить. Для этого достаточно предусмотреть, чтобы определение атрибута CustomerID находилось после определения атрибута OrderCount.

Дополнительные возможности применения символов @ и /

Как уже было сказано выше, язык XPath сам по себе является достаточно сложным, и для его описания нужна отдельная книга, но автор хочет привести еще хотя бы некоторый объем информации, дополняющий предыдущее описание.

Символы @ и / открывают широкий спектр возможностей, позволяющих формировать выходные данные в коде XML в полном соответствии с поставленными требованиями, поэтому конструкции, основанные на их использовании, по-видимому, соответствуют потребностям большинства приложений начального уровня. Если же требуется применение более сложных средств представления данных, то можно воспользоваться некоторыми описанными ниже возможностями.

- Объединение данных по такому же принципу, который основан на использовании подстановочных символов, что позволяет представить всю выходную информацию в целом как текстовые данные, не требующие разделения на отдельные столбцы.

- Внедрение собственных данных XML, полученных из столбцов с типами данных XML.
- Применение средств проверки узлов языка XPath; таковыми являются специальные директивы XPath, позволяющие уточнять способы обработки данных.
- Использование директивы `data()` для обеспечения возможности представления в коде XML нескольких значений в виде одного многозначного значения.
- Использование пространств имен.

Функция OPENXML

Значительная часть настоящей главы посвящена описанию методов преобразования реляционных данных в данные, представленные в коде XML. Но из приведенного описания следует также вполне обоснованный вывод, что в СУБД SQL Server должна быть также предусмотрена возможность открыть строку кода XML и преобразовать содержащиеся в ней данные в табличный формат, подходящий для обработки с помощью языка SQL. И действительно, такие функциональные возможности введены в последней версии СУБД SQL Server для обеспечения возможности выполнения действий, обратных по отношению к реализуемым с помощью конструкции `FOR XML`, о чем уже было сказано выше.

`OPENXML` — это функция для работы с наборами строк, которая позволяет открыть переданную ей строку во многом на основе такого же принципа, по которому действуют другие функции для работы с наборами строк (такие как `OPENQUERY` и `OPENROWSET`). Это означает, что документ XML можно использовать в операции соединения или даже сделать его источником входных данных, применив операцию `INSERT...SELECT` или `SELECT INTO`. При этом основное различие между `OPENXML` и другими функциями состоит в том, что в сочетании с этой функцией необходимо использовать целый ряд системных хранимых процедур для подготовки документа, а затем для очистки памяти после завершения обработки документа.

Для подготовки документа к работе служит процедура `sp_xml_preparedocument`, которая перемещает переданную ей строку в память и выполняет предварительный синтаксический анализ этой строки для достижения оптимальной производительности запроса. Документ XML остается в памяти до тех пор, пока пользователь явно не укажет, что требуется удалить этот документ, или пока не завершится использование соединения, с помощью которого была вызвана процедура `sp_xml_preparedocument`. Вызов процедуры `sp_xml_preparedocument` имеет следующий простой синтаксис:

```
sp_xml_preparedocument @hdoc = <integer variable> OUTPUT,
[, @xmltext = <character data>]
[, @xpath_namespaces = <url to a namespace>]
```

> Следует отметить, что если должно быть задано значение URL, указывающее на пространство имен `<url to a namespace>`, то заданное значение должно быть заключено с обоих концов в символы < и > (например, `<root xmlns:sql="run:schemas-microsoft-com:xml-sql">`).

Параметры этой хранимой процедуры, как описано ниже, в основном не требуют пояснений.

- `@hdoc`. Программисты, которым приходилось использовать в своей работе API-интерфейс Windows (или какие-то другие системы программирования, но приведенный пример является наиболее характерным), по-видимому, знают, какой смысл имеет префикс "h"; он представляет собой обозначение дескриптора в венгерской записи. Дескриптор по существу представляет собой указатель на блок памяти, в котором хранятся некоторые данные (причем фактически разнообразие представленных в этом блоке данных является почти неограниченным). В данном случае параметр `@hdoc` представляет собой дескриптор документа XML, который передается в СУБД SQL Server для последующего синтаксического анализа и сохранения. Это — выходная переменная, а это означает, что переменная, на которую здесь задана ссылка, после возврата управления из хранимой процедуры будут содержать дескриптор документа XML. Такие данные следует обязательно сохранить, поскольку они потребуются для дальнейшей работы с функцией `OPENXML`.
- `@xmltext`. Назначение этого параметра соответствует его имени — он указывает на действительный текст XML, который должен быть подвергнут синтаксическому анализу и предназначен для использования в дальнейшей работе.
- `@xpath_namespaces`. Ссылка (ссылки) на пространство имен, которая требуется для обеспечения правильной обработки кода XML.

После вызова этой хранимой процедуры и сохранения дескриптора документа можно приступить к использованию функции `OPENXML`. Синтаксис вызова указанной функции является немного более сложным:

```
OPENXML(<handle>,<XPath to base node>[, <mapping flags>])
[WITH (<Schema Declaration>|<Table Name>)]
```

Выше в данной главе уже было посвящено несколько слов описанию параметров, содержащих значение дескриптора, таких как `<handle>`; вообще говоря, дескриптор представляет собой целочисленное значение, которое должно быть получено в виде выходного параметра после вызова процедуры `sp_xml_preparedocument`.

При вызове функции `OPENXML` должен быть задан параметр `<XPath to base node>`, представляющий собой обозначение пути к узлу, который должен служить в качестве начальной точки для всех запросов. С другой стороны, параметр `<Schema Declaration>` позволяет ссылаться на любые части документа XML с помощью указания пути перемещения относительно базового узла, заданного в этом параметре.

Следующим параметром функции `OPENXML` являются флажки отображения, `<mapping flags>`. Эти флажки позволяют указать, должно ли быть в результатах вызова функции `OPENXML` отдано предпочтение элементам или атрибутам. Возможные значения флажков отображения и их описания приведены в табл. 16.4.

Наконец, должна быть указана схема или таблица. Для тех, кто не знаком с языком XPath, задача определения схемы может оказаться довольно трудоемкой. Но, к счастью, применяемые при этом конструкции XPath являются не слишком сложными, поэтому процедура составления схемы может быть довольно быстро доведена буквально до автоматизма (используемые при этом конструкции во многом напоминают определения каталогов в файловой системе Windows).

Таблица 16.4. Возможные значения флажков отображения и их описания

Десятичное значение	Описание
0	Это значение, как и описанное ниже значение 1, позволяет указать, что должны использоваться только атрибуты. Но в отличие от 1, данное значение не может применяться в сочетании с 2 или 8 (поскольку, например, 0 + 2 = 2). Значение 0 задано по умолчанию
1	Если это значение не задано в сочетании с описанным ниже значением 2, то используются только атрибуты. Если атрибут с указанным именем отсутствует, то происходит возврат NULL-значения. Значение 1 может также применяться в сумме со значением 2 или 8 (или с обоими этими значениями) в целях осуществления действий, предусмотренных всеми заданными опциями, но в первую очередь выполняется действие, предусмотренное значением 1. Если с помощью конструкции XPath обнаруживаются и атрибут, и элемент с одинаковыми именами, то предпочтение отдается атрибуту
2	Если это значение не задано в сочетании с описанным выше значением 1, то используются только элементы; если элемент с указанным именем отсутствует, то возвращается NULL-значение. Значение 2 может также применяться в сумме со значением 1 или 8 (или с обоими этими значениями) для осуществления действий, предусмотренных всеми заданными опциями. Но если задано также значение 1, то осуществляется отображение атрибута, при условии, что он существует, а если атрибут не существует, то используется элемент; если не существует и элемент, то происходит возврат NULL-значения
8	Это значение может применяться в сочетании с описанными выше значениями 1 или 2. Для хранения полученных данных не следует привлекать вспомогательное свойство `@mp:xmltext` (поскольку в таком случае для выборки данных придется использовать элемент схемы `MetaProperty`). Если для обработки документа не предусматривается применение метасвойств (а необходимость в этом чаще всего отсутствует), то можно порекомендовать эту опцию, поскольку она позволяет немного (возможно, даже очень немного) сократить издержки при выполнении обработки

Сама схема может немного изменяться в зависимости от способа ее объявления. В качестве объявления схемы применяется примерно такое определение:

```
WITH (
<Column Name> <data type> [{<Column XPath>|<MetaProperty>}]
[,<Column Name> <data type> [{<Column XPath>|<MetaProperty>}]
 ...
```

Описание этого определения приведено ниже.

- Параметр с обозначением имени столбца, `<Column Name>`, полностью соответствует своему названию — он представляет собой имя атрибута или элемента, применяемого для выборки данных. Этот параметр служит также в качестве имени, на которое можно сослаться при формировании списка выборки, выполнении операции соединения и т.д.
- Параметр с обозначением типа данных, `<data type>`, представляет собой любой допустимый тип данных SQL Server. Поскольку в документе XML могут быть заданы типы данных, не имеющие эквивалентов в СУБД SQL Server, в случае необходимости автоматически осуществляется приведение типа, но обычно это влечет за собой получение предсказуемых результатов.
- Параметр `<Column XPath>` обозначает шаблон XPath (заданный относительно узла, указанного в качестве начальной точки для функции `OPENXML`). Шаблон XPath позволяет получить доступ к узлу, данные которого требуются для формирования столбца. Должен ли при этом использоваться элемент или атрибут,

зависит от описанного выше параметра с обозначением флажков отображения. Если параметр <Column XPath> не задан, то СУБД SQL Server действует на основании предположения, что должны быть получены данные текущего узла, согласно определению начальной точки для оператора вызова функции OPENXML.

- Параметр <MetaProperty> задает множество специальных переменных (метасвойств), на которые можно ссылаться в запросах OPENXML. Эти переменные определяют различные аспекты применения той или иной части модели DOM документа XML, которые интересуют пользователя. Чтобы ввести в действие указанные переменные, достаточно заключить их в одинарные кавычки и поместить в соответствующее место в значении параметра <Column XPath>. Перечень доступных метасвойств приведен ниже.
- @mp:id. Переменную @mp:id не следует путать с директивой id, применяемой в сочетании с конструкцией FOR XML, которая рассматривалась при описании опции EXPLICIT. Дело в том, что свойство @mp:id имеет аналогичное назначение, но представляет собой уникальный идентификатор узла DOM (в области определения документа). Еще одной отличительной особенностью свойства @mp:id является то, что его значение формируется системой, поэтому при его использовании можно быть уверенным, что требуемое значение уже задано. Кроме того, гарантируется, что данное свойство продолжает хранить ссылку на один и тот же узел модели DOM документа XML до тех пор, пока этот документ остается в памяти. Если значение переменной @mp:id равно нулю, то узел, к которому она относится, представляет собой корневой узел (свойство @mp:parentid этого узла, о котором речь пойдет ниже, составляет NULL).
- @mp:parentid. Это свойство соответствует описанному выше свойству @mp:id, но относится только к родительскому узлу.
- @mp:localname. Свойство @mp:localname определяет часть полностью уточненного имени узла, соответствующую локальному имени. Это свойство используется в сочетании с префиксом и универсальным идентификатором ресурса (Uniform Resource Identifier – URI) пространства имен (обычно идентификатор URI пространства имен начинается с префикса URN) для составления имен узлов элементов или узлов атрибутов.
- @mp:parentlocalname. Это свойство соответствует описанному выше свойству @mp:localname, но относится только к родительскому узлу.
- @mp:namespaceuri. Свойство @mp:namespaceuri позволяет определить идентификатор URI пространства имен текущего элемента. Если это свойство имеет NULL-значение, то пространство имен не определено.
- @mp:parentnamespaceuri. Это свойство соответствует описанному выше свойству @mp:namespaceuri, но относится только к родительскому узлу.
- @mp:prefix. Свойство @mp:prefix содержит префикс пространства имен для имени текущего элемента.
- @mp:prev. Свойство @mp:prev содержит значение свойства @mp:id предыдущего узла, сестринского по отношению к текущему узлу. С помощью этого свойства можно получить информацию об упорядочении элементов на текущем уровне иерархии. Например, если значение свойства @mp:prev составляет NULL, то рассматриваемый узел является первым по порядку узлом на данном уровне дерева.

- `@mp:xmltext`. Метасвойство `@mp:xmltext` используется для обработки данных и содержит действительный код XML текущего элемента.

Безусловно, разработчик всегда имеет возможность избавиться от огромного объема работы, постаравшись обойтись без использования всех указанных параметров. Этой цели можно достичь, подготовив таблицу, которая устанавливает непосредственную связь (определяемую по именам и типам данных) с начальной точкой XPath, заданной в функции `OPENXML`. Подготовив такую таблицу, следует указать ее имя, после чего все необходимые преобразования будут выполнены в СУБД SQL Server автоматически!

Таким образом, действия по подготовке документа XML являются достаточно трудоемкими, но на этом работа не заканчивается. Дело в том, что после завершения подготовки документа XML необходимо вызвать процедуру `sp_xml_removedocument`, чтобы освободить память, в которой хранился документ XML. Но, к счастью, оператор вызова этой процедуры имеет очень простой синтаксис:

```
sp_xml_removedocument [hdoc = ]<handle of XML doc>
```

Следует отметить, что для обеспечения бесперебойного функционирования приложения необходимо всегда стремиться своевременно уничтожать ненужные объекты. Безусловно, уничтожение многих объектов, ставших ненужными, происходит автоматически. В частности, функции своевременного освобождения ресурсов осуществляются и в СУБД SQL Server, но не следует рассчитывать на то, что СУБД сможет автоматически освободить все ресурсы. Например, СУБД SQL Server удаляет из оперативной памяти ненужные объекты после завершения соединения, но если используется пул соединений, выполнение такой операции становится затруднительным. А если система постоянно работает в условиях высокой нагрузки, то некоторые неиспользуемые соединения могут продолжать занимать отведенные им ресурсы. Но для разработчика совсем несложно предусмотреть своевременный вызов хранимой процедуры для освобождения ресурсов, поэтому рекомендуется обязательно удалять все ненужные объекты.

Итак, мы в основном рассмотрели все, что необходимо для решения реальной задачи, поэтому приступим к изучению достаточно сложного примера.

Допустим, что происходит слияние двух компаний, поэтому часть данных одной из компаний необходимо передать в базу данных другой компании. Предположим, что необходимо импортировать данные о некоторых поставщиках, информация о которых отсутствует в базе данных компании Northwind. Ниже приведен сценарий, который может служить образцом того, как данные импортируются из документа XML.

```
USE Northwind
DECLARE @idoc     int
DECLARE @xmldoc   nvarchar(4000)
-- Определить документ XML
SET @xmldoc = '
<ROOT>
<Shipper ShipperID="100" CompanyName="Billy Bob&apos;s Pretty Good Shipping"/>
<Shipper ShipperID="101" CompanyName="Fred&apos;s Freight"/>
</ROOT>
'
-- Загрузить документ XML и выполнить его синтаксический анализ
-- в оперативной памяти
EXEC sp_xml_preparedocument @idoc OUTPUT, @xmldoc
```

```
-- Рассмотрим, как выглядит таблица Shippers до вставки
SELECT * FROM Shippers
-- ShipperID представляет собой столбец идентификации, поэтому необходимо
-- разрешить непосредственные обновления
SET IDENTITY_INSERT Shippers ON
-- Рассмотрим данные XML в табличном формате
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
   ShipperID        int,
 CompanyName        nvarchar(40))
-- Выполнение вставки с учетом имеющихся данных
INSERT INTO Shippers
(ShipperID, CompanyName)
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
   ShipperID        int,
 CompanyName        nvarchar(40))
-- Восстановление обычных значений параметров
SET IDENTITY_INSERT Shippers OFF
-- А теперь рассмотрим, как выглядит таблица Shippers после вставки
SELECT * FROM Shippers
-- После этого можно удалить документ XML из памяти
EXEC sp_xml_removedocument @idoc
```

Полученный при этом результат действительно отвечает предъявленным требованиям.

```
ShipperID      CompanyName                          Phone
-------------------------------------------------------------------------
1              Speedy Express                       (503) 555-9831
2              United Package                       (503) 555-3199
3              Federal Shipping                     (503) 555-9931
4              Speedy Shippers, Inc.                (503) 555-5566
5              Speedy Shippers, Inc                 NULL
100            Billy Bob's Pretty Good Shipping     NULL
101            Fred's Freight                       NULL
102            Readyship                            (503)555-1234
103            MyShipper                            (503)555-3443
```

Таким образом, мы успешно справились с решением простой задачи, а теперь перейдем к рассмотрению на практике некоторых дополнительных опций `OPENXML`.

На этот раз воспользуемся частью фрагмента XML, который был сформирован с помощью конструкции `FOR XML EXPLICIT` выше в настоящей главе. Но в данном примере выполним выборку данных `ProductID`, `CategoryID`, `OrderID` и `OrderDate`. А чтобы немного усложнить задачу, попытаемся также осуществить выборку свойства `@mp:prev`.

Прежде чем приступить к описанию кода, приведенного ниже, напомним, что документы XML без корневого элемента не рассматриваются как формально правильные. Но в СУБД SQL Server не предусмотрено автоматическое введение корневых элементов в результаты, полученные с помощью запросов FOR XML, поскольку после введения корневого элемента исключается возможность объединять в одном документе результаты нескольких запросов. Поэтому корневой элемент должен быть введен вручную перед передачей документа на обработку в процедуру `sp_xml_preparedocument`.

```
DECLARE @idoc int
DECLARE @doc varchar(4000)
-- Код XML, полученный с применением конструкции FOR XML EXPLICIT
SET @doc ='
<root>
<Product ProductID="1" CategoryID="1" />
<Product ProductID="2" CategoryID="1">
 <Order OrderID="10813" OrderDate="1998-01-05T00:00:00" />
</Product>
<Product ProductID="24" CategoryID="1" />
<Product ProductID="34" CategoryID="1" />
<Product ProductID="35" CategoryID="1" />
<Product ProductID="38" CategoryID="1">
 <Order OrderID="10816" OrderDate="1998-01-06T00:00:00" />
 <Order OrderID="10817" OrderDate="1998-01-06T00:00:00" />
</Product>
<Product ProductID="39" CategoryID="1" />
<Product ProductID="43" CategoryID="1">
 <Order OrderID="10814" OrderDate="1998-01-05T00:00:00" />
 <Order OrderID="10819" OrderDate="1998-01-07T00:00:00" />
</Product>
<Product ProductID="67" CategoryID="1" />
<Product ProductID="70" CategoryID="1">
 <Order OrderID="10810" OrderDate="1998-01-01T00:00:00" />
</Product>
</root>
'
-- Создание внутреннего представления документа XML
EXEC sp_xml_preparedocument @idoc OUTPUT, @doc
-- Выполнение оператора SELECT с применением средств формирования
-- наборов строк OPENXML
SELECT *
FROM OPENXML (@idoc, '/root/Product/Order', 1)
     WITH (ProductID int '../@ProductID',
          Category int '../@CategoryID',
          OrderID int '@OrderID',
          OrderDate varchar(19) '@OrderDate',
          Previous varchar(10) '@mp:prev')
EXEC sp_xml_removedocument @idoc
```

В этом коде заслуживает внимания то, что при формировании метаданных используются относительные обозначения путей в ссылках XPath. Сравните структуру этих обозначений с иерархией элементов в документе XML. Даже поверхностный анализ показывает, что структура обозначений путей XPath весьма напоминает структуру каталогов операционных систем DOS и Windows.

Ниже приведены результаты выполнения рассматриваемого сложного сценария.

```
ProductID   Category    OrderID     OrderDate             Previous
----------  ----------  ---------   -------------------   --------------
2           1           10813       1998-01-05T00:00:00   NULL
38          1           10816       1998-01-06T00:00:00   NULL
38          1           10817       1998-01-06T00:00:00   23
43          1           10814       1998-01-05T00:00:00   NULL
43          1           10819       1998-01-07T00:00:00   35
70          1           10810       1998-01-01T00:00:00   NULL
```

Вполне очевидно, что в данном случае мы сумели выполнить выборку данных XML на различных уровнях иерархии. Как оказалось, язык XPath позволяет перейти в любую точку иерархического дерева XML, в которой должна быть осуществлена выборка данных.

Таким образом, выше в данной главе описан способ, позволяющий получить результаты запросов XML, а также способ, с помощью которого данные, представленные в документе XML, могут быть преобразованы в реляционные данные. В следующем разделе представлены еще более интересные сведения и описаны средства взаимодействия с функциональными возможностями поддержки языка XML, которые предусмотрены в СУБД SQL Server.

Краткое описание преобразований XSL

В настоящем разделе рассматривается последняя, но наиболее сложная тема из всех, которые быть представлены в данной главе.

Преобразования XSL (Extensible Stylesheet Language — расширяемый язык стилей) относятся к числу средств, позволяющих преобразовывать документы XML в другие формы.

Чтобы можно было проще приступить к изложению краткого вводного описания, рассмотрим следующий документ XML, полученный на основании данных из базы данных Northwind:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
 <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
       <Products ProductID="28" ProductName="Rossle Sauerkraut"/>
       <Products ProductID="39" ProductName="Chartreuse verte"/>
       <Products ProductID="46" ProductName="Spegesild"/>
 </Customer>
 <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils">
       <Products ProductID="28" ProductName="Rossle Sauerkraut"/>
       <Products ProductID="29" ProductName="Thuringer Rostbratwurst"/>
       <Products ProductID="31" ProductName="Gorgonzola Telino"/>
       <Products ProductID="38" ProductName="Cote de Blaye"/>
       <Products ProductID="39" ProductName="Chartreuse verte"/>
       <Products ProductID="41" ProductName="Jack&amp;apos;s New England Clam Chowder"/>
       <Products ProductID="46" ProductName="Spegesild"/>
       <Products ProductID="49" ProductName="Maxilaku"/>
 </Customer>
</root>
```

В этом документе наглядно показано одно из наиболее привлекательных свойств языка XML — возможность представления с его помощью иерархических данных. В рассматриваемом случае мы сталкиваемся с ситуацией, в которой заказчики заказывают различные товары. А приведенный документ XML позволяет легко определить, какие товары приобретены заказчиками. Но возникает резонный вопрос: можно ли использовать этот документ XML для проведения дополнительного анализа?

В частности, было бы любопытно определить, какими заказчиками был заказан каждый товар. В связи с постановкой этого вопроса подход к организации структуры данных существенно изменяется. Выполнить это задание было бы гораздо проще, если бы в документе данные о товарах были представлены элементами верхнего уровня, а

данные о заказчиках – элементами нижнего уровня. Безусловно, при использовании такой структуры данных в документе много раз упоминались бы одни и те же заказчики (а не товары), поскольку эти данные были бы заданы в связи с указанием каждого товара, но благодаря этому было бы проще найти ответ на рассматриваемый вопрос.

При использовании языка XML в сочетании со средствами преобразования XSL (XSL Transformations – XSLT) любое подобное преобразование структуры становится несложным. При этом не приходится даже модифицировать исходную структуру данных. Достаточно лишь применить к документу XML определенное преобразование, для того чтобы он выглядел иначе.

Когда речь идет о каком-либо представлении данных, под этим не подразумевается подготовка данных для визуального просмотра, поскольку само понятие представления данных включает в себя намного больше, чем просто выбор способа подготовки данных для восприятия людьми. Скорее всего, под представлением данных следует понимать их подготовку для использования в какой-то конкретной форме. Например, если данные о заказчиках и заказанных ими товарах подготовлены так, что на вершине иерархии данных находится информация о заказчиках, то используемое при этом представление мало подходит для получения ответов на вопросы, в большей степени касающиеся товаров. Чтобы проще было находить ответы на вопросы о товарах, необходимо изменить структуру данных таким образом, чтобы на вершине иерархии находилась информация о заказах, как и в рассматриваемом случае.

Поэтому рассмотрим те же данные, но преобразованные в другое представление:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
 <Products ProductID="28" ProductName="Rossle Sauerkraut">
       <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
       <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils"/>
 </Products>
 <Products ProductID="29" ProductName="Thuringer Rostbratwurst">
       <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils"/>
 </Products>
 <Products ProductID="31" ProductName="Gorgonzola Telino">
       <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils"/>
 </Products>
 <Products ProductID="38" ProductName="Cote de Blaye">
       <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils"/>
 </Products>
 <Products ProductID="39" ProductName="Chartreuse verte">
       <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
       <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils"/>
 </Products>
 <Products ProductID="41" ProductName="Jack&amp;apos;s New England Clam Chowder">
       <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils"/>
 </Products>
 <Products ProductID="46" ProductName="Spegesild">
       <Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste"/>
       <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils"/>
 </Products>
 <Products ProductID="49" ProductName="Maxilaku">
       <Customer CustomerID="BLONP" CompanyName="Blondesddsl pere et fils"/>
 </Products>
</root>
```

Вполне очевидно, что данные остались прежними, но теперь они могут рассматриваться с другой точки зрения.

Но возможности преобразований XSL выходят за рамки их использования исключительно для модификации форм представления XML. Безусловно, описание преобразований всех прочих типов, которые могут быть осуществлены с помощью языка XSL, не относится к тематике данной книги, но важно отметить, что в качестве исходных и результирующих форматов для преобразований XSL могут быть заданы весьма разнообразные определения. Обычно принято считать, что язык XSL предназначен для преобразования документов XML из одной компоновки в другую, но фактически с его помощью могут также осуществляться преобразования в совершенно другие форматы, такие как файлы CSV или даже документы Word.

Резюме

В настоящей главе представлено чрезвычайно краткое описание способов применения языка XML, определений DTD, схем, а также языка XPath в СУБД SQL Server. Очевидно, что в одной главе невозможно раскрыть такую широкую тему, как язык XML и связанные с ним технологии, но я надеюсь, что эта глава позволила вам понять назначение языка XML и получить сведения, позволяющие использовать его хотя бы в минимальном объеме.

Следует еще раз подчеркнуть, что язык XML стал основой наиболее важных технологических достижений в области обработки информации за последние десять с лишним лет, поскольку с применением этого языка успешно создаются чрезвычайно разносторонние способы описания данных, которые могут быть легко приспособлены к конкретным условиям. Благодаря этому не только значительно упрощается разработка Web-узлов, но и появляется возможность обеспечить бесперебойный обмен информацией между деловыми предприятиями, а эта задача действительно оставалась невыполненной в течение очень долгого времени.

17

Общее описание средств формирования отчетов

После написания всех запросов и выполнения всех хранимых процедур остается решить не менее важную задачу, позволяющую упростить использование полученных данных и предоставить к ним доступ конечным пользователям, — оформить отчет.

На первый взгляд формирование отчетов может показаться одной из самых простых задач, но в действительности для создания качественных отчетов иногда требуются значительные усилия. При этом невозможно ограничиться просто выводом ничего не значащих данных; эти данные должны быть осмысленными, кроме того, по возможности должны помочь сосредоточить внимание того лица, которому предоставляется отчет, на нужной информации. Ниже приведены рекомендации по подготовке таких отчетов, которые действительно будут использоваться и поэтому оцениваться как полезные.

- Отчеты должны включать строго выверенное количество данных. Не следует пытаться вместить в один отчет слишком много информации, в то же время отчет не должен быть малосодержательным. Пользователи обычно очень быстро теряют интерес к отчету, представляющему собой лабиринт, в котором трудно разобраться, и, как правило, обнаруживается, что такой отчет больше не применяется после первых нескольких просмотров. Аналогичным образом, отчет, почти не содержащий полезной информации, лишь наспех просматривают и отбрасывают без каких-либо глубоких размышлений. При составлении отчета необходимо найти равновесие и ввести в него наиболее приемлемый объем самых нужных данных.
- Отчет должен быть привлекательным. К сожалению, многие этого не учитывают, но при подготовке отчета необходимо помнить о том, что он должен быть качественно оформлен, иными словами, необходимо добиться того, чтобы отчет был аккуратным и вызывал интерес. Если отчет выглядит малопривлекательно, то с ним трудно работать.

В настоящей главе кратко рассматриваются инструментальные средства формирования отчетов, Reporting Services, которые впервые появились в версии SQL Server 2005. Как и в отношении многих других дополнительных средств SQL Server, описанных в данной книге, приведенное описание в большей степени позволяет лишь ознакомиться с основными возможностями, поскольку объем материала слишком велик для того, чтобы можно было поместить все необходимые сведения в одной главе даже более объемной книги. А если вы обнаружите, что служба Reporting Services соответствует вашим потребностям, то изложенные здесь сведения позволят проще приступить к изучению книги, специально посвященной описанию службы Reporting Services, такой как *Professional SQL Server Reporting Services.*

Краткое описание службы Reporting Services

Заниматься формированием отчетов приходится почти всем программистам. Иногда такие отчеты представляют собой распечатки, полученные с помощью принтера (и, возможно, составленные с помощью таких простейших средств, как функции формирования отчетов СУБД Access; хотя, по мнению автора, эти функции — одно из лучших программных средств Access). Для подготовки других отчетов может применяться достаточно мощная машина формирования отчетов, такая как Crystal Reports. Но даже если вы не используете эти удобные программы формирования отчетов, то, по мнению многих, не следует ограничиваться применением для получения распечатки данных только теми операторами вывода, которые могут быть включены в хранимую процедуру, поскольку при этом приходится довольствоваться получением чрезвычайно простых (и не слишком привлекательных) отчетов.

В действительности в наши дни уже недопустимо ограничиваться такими простейшими отчетами. А когда возникает необходимость обеспечить более сложное форматирование вывода, на помощь приходят службы Reporting Services. Фактически службы Reporting Services могут эксплуатироваться в двух описанных ниже режимах работы.

- Формирование отчетов с помощью моделей отчетов. В этом режиме используется относительно простой Web-интерфейс, позволяющий конечным пользователям самостоятельно создавать собственные простые отчеты.
- Формирование отчетов с помощью программы Visual Studio. В этом режиме могут быть созданы весьма сложные отчеты, и для этого может даже не потребоваться написание кода (фактически для создания простых отчетов могут использоваться функциональные средства перетаскивания и фиксации (drag and drop), которые рассматриваются в качестве примеров в данной главе).

Следует отметить, что в конечном итоге пользователи получают доступ к отчетам, созданным с помощью служб Reporting Services, через тот же Web-узел, который применялся разработчиком, но для предоставления доступа пользователям применяются совсем другие сетевые программные средства (кроме того, как будет указано в этой главе, создание отчетов осуществляется иначе, чем их просмотр).

Кроме того, службы Reporting Services предоставляют возможность предварительного формирования отчетов (это удобно, если для выполнения запросов, предоставляющих данные в отчет, требуется много времени), а также распространения отчетов по электронной почте.

Создание простых моделей отчетов

Чтобы приступить к созданию простой модели, вначале откройте программу Business Intelligence Studio.

Следует отметить, что эта программа полностью отличается от программы SQL Server Management Studio, с которой мы работали до сих пор. Отличительной особенностью рабочей области программы Business Intelligence Studio является то, что данная рабочая область главным образом предназначена для использования разработчиками (а не администраторами), а также позволяет получить доступ ко многим дополнительным службам, предоставляемым в программном обеспечении SQL Server, кроме той основной реляционной машины, которая главным образом рассматривалась до сих пор в настоящей книге. В текущей главе не только показано, как работать с программой Business Intelligence Studio, но и кратко описаны те задачи, которые будут решены с помощью службы Integration Services в главе 18.

Выберите команды File⇨New Project, чтобы открыть диалоговое окно New Project (рис. 17.1).

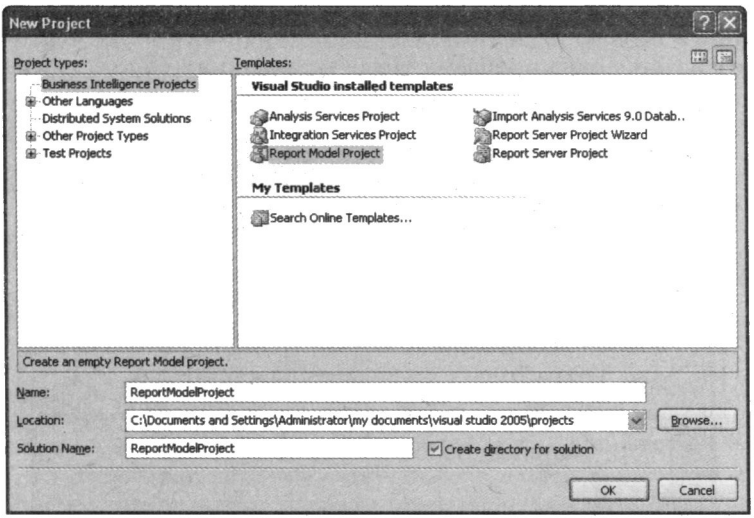


Рис. 17.1. Диалоговое окно New Project

Выберите для обозначения модели отчета подходящее имя (в данном примере используется довольно выразительное имя `ReportModelProject`) и щелкните на кнопке OK. В результате вы получите доступ к среде разработки Visual Studio.

Следует отметить, что вид этого диалогового окна может оказаться немного иным, в зависимости от того, была ли инсталлирована программа Visual Studio, а также, если это сделано, от состава конкретных инсталлированных языков и шаблонов. На рис. 17.1 показан вид окна, который оно приобретает при использовании наиболее универсального способа инсталляции, который предусматривает установку только программного обеспечения SQL Server.

Если все еще применяется заданная по умолчанию конфигурация среды Visual Studio, то окно Solution Explorer должно находиться в правой верхней части. Чтобы определить источник данных, щелкните правой кнопкой мыши на обозначении Data Sources и выберите команду Add New Data Source (рис. 17.2).

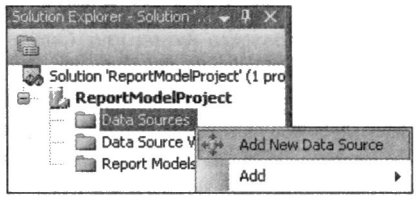


Рис. 17.2. Вызов на выполнение команды Add New Data Source

В результате, по всей видимости (если вы еще не открывали это окно и не устанавливали флажок Don't show me this page again – (Больше не открывать эту страницу)), появится диалоговое окно с приветствием, Welcome, программы-мастера Data Source Wizard. Щелкните на кнопке Next, чтобы приступить к работе и воспользоваться страницей выбора источника данных программы Data Source Wizard. Тем не менее обнаруживается одна проблема – еще не задано ни одного источника данных (рис. 17.3).

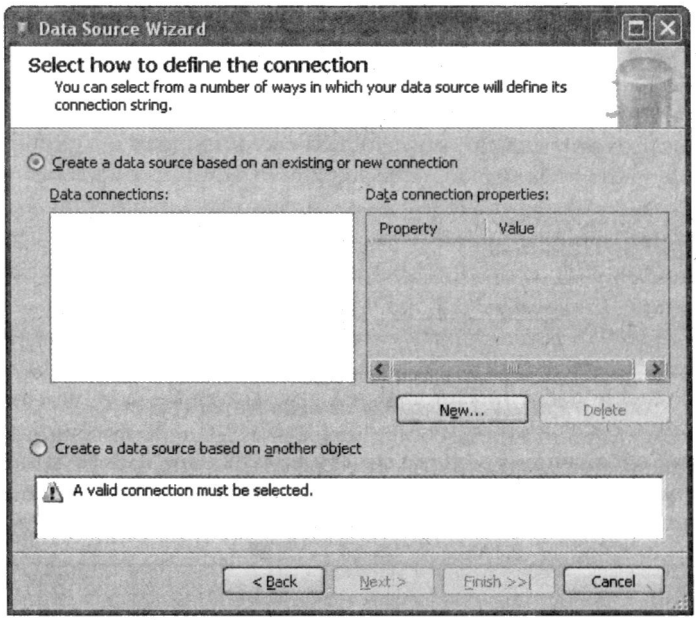


Рис. 17.3. Окно Data Source Wizard

По-видимому, не стоит и говорить о том, что если в поле Data connections отсутствуют определения каких-либо соединений с источниками данных, одно из которых мы могли бы выбрать, остается только щелкнуть на кнопке New, чтобы создать новое соединение.

Когда я впервые увидел это очередное диалоговое окно, я был немного удивлен, обнаружив то, что в данном случае открывается не такое диалоговое окно определения нового соединения, которое неоднократно использовалось в программе Management Studio; правда, в нем содержатся те же основные элементы, хотя и с немного другой визуальной компоновкой (короче говоря, не следует задумываться над тем, что это окно выглядит немного иначе).

После этого откроется окно Connection Manager (рис. 17.4).

***Рис. 17.4. Окно** Connection Manager*

Окно Connection Manager по составу своих элементов напоминает окна, которые уже рассматривались в некоторых главах настоящей книги, но в нем имеется также и несколько новинок, поэтому кратко рассмотрим некоторые его основные элементы.

- Server name. Содержимое этого поля, в котором должно быть указано имя сервера, соответствует его названию, а для заполнения этого поля применяются такие же правила, как и в других подобных случаях, описанных в данной книге. Укажите сервер, к которому вы хотите подключиться в этом соединении, а если требуется подключиться к применяемому по умолчанию экземпляру SQL Server, который относится к локальному серверу, можно также воспользоваться псевдонимом "`(local)`" или "`.`" (точка).
- Use Windows Authentication или Use SQL Server Authentication. Выберите тип аутентификации, используемый для подключения к серверу. Корпорация Microsoft (а также сам автор) рекомендует использовать тип Windows Authentication, но если сервер не входит в состав вашего домена или админи-

стратор не предоставил права доступа непосредственно вашей регистрационной записи Windows, то можно применить предоставленные администратором имя пользователя и пароль, относящиеся непосредственно к СУБД SQL Server (напомним, что в некоторых примерах данной книги использовались имя пользователя и пароль `MyLogin` и `MyPassword`).

- Connect to a database. Для заполнения области окна, обозначенной как Connect to a database, потребуется больше внимания. В частности, необходимо либо продолжить логическую последовательность выбора одной из баз данных на указанном перед этим сервере, либо применить вариант с подключением непосредственно к файлу `mdf` (в этом случае преобразование имени файла в имя базы данных осуществляется машиной SQL Server Express).

В окне, показанном на рис. 17.4, выбран локальный сервер (поскольку в качестве имени сервера задана точка), а для использования в качестве базы данных выбрана привычная база данных Northwind.

После заполнения необходимых полей в окне Connection Manager щелкните на кнопке OK. Появится диалоговое окно программы Data Source Wizard, отличное от того, которое было показано впервые (рис. 17.5).

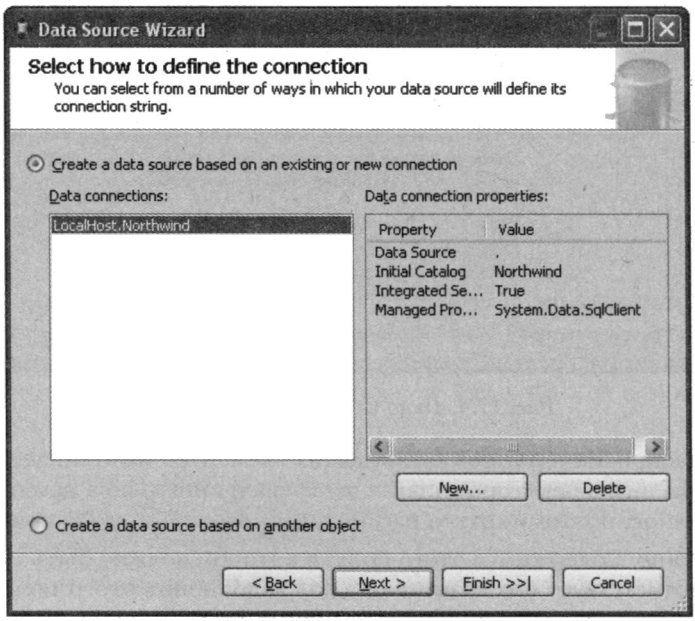


Рис. 17.5. Диалоговое окно программы Data Source Wizard

В примере, показанном на рис. 17.5, демонстрируется только одно соединение, созданное на предыдущем этапе, но можно было бы создать несколько соединений, а затем выбирать между ними. В этом окне заслуживает также внимания таблица свойств соединения с источником данных, находящаяся в правой части диалогового окна.

Щелкните на кнопке Next, чтобы открыть последнее диалоговое окно программы-мастера (рис. 17.6).

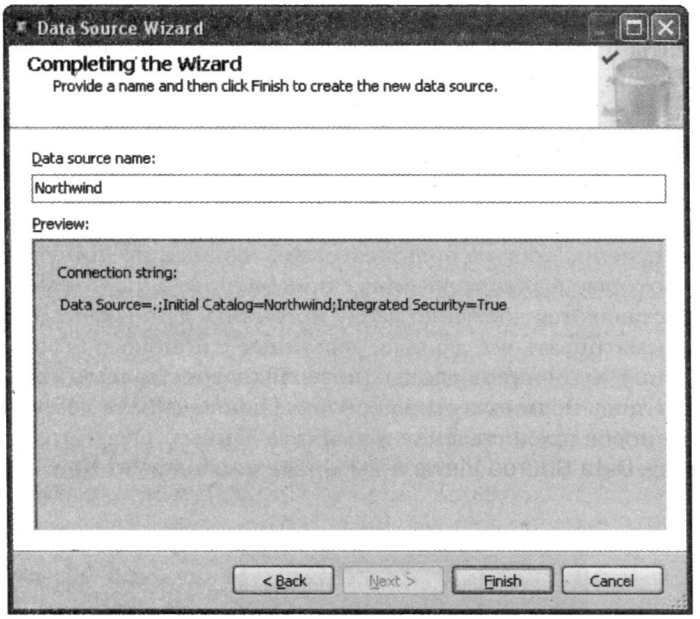


Рис. 17.6. Последнее окно программы Data Source Wizard

Обратите внимание на то, что в этом окне в качестве имени источника данных, Data source name, по умолчанию задано имя выбранной базы данных, но в действительности это имя относится к источнику данных. Такой источник данных может быть выбран в любое время, когда это потребуется, но в результате такого выбора будет по-прежнему осуществляться подключение к базе данных `Northwind` на локальном сервере.

Следует также отметить, что на рис. 17.6 показана строка соединения, сформированная автоматически. Применение строк соединения лежит в основе всех современных методов обеспечения связи с базой данных, поскольку строки соединения используются на определенном уровне практически в любой модели обеспечения связи (например, в провайдерах данных, действующих под управлением инфраструктуры .NET, а также в интерфейсах OLEDB и ODBC). Если бы на предыдущем этапе подключения к источнику данных применялись другие опции (в частности, если бы были заданы имя пользователя и пароль SQL Server вместо опций защиты Windows), то в строке соединения использовались бы немного другие параметры, а также, безусловно, передавались бы некоторые другие значения.

После завершения работы с окном (см. рис. 17.6) щелкните на кнопке Finish, и в проекте появится новый источник данных (рис. 17.7).

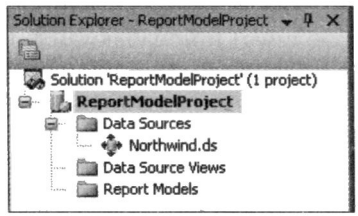


Рис. 17.7. Новый источник данных, появившийся в окне Solution Explorer

В модели ReportModelProject предусмотрены не только источники данных, но и другие компоненты (см. рис. 17.7), поэтому после определения источника данных необходимо перейти на следующий этап и создать представление источника данных.

Представление источника данных

Представления источников данных немного напоминают обычные представления, о которых шла речь в главе 10. В частности, представления источников данных могут служить для обеспечения доступа пользователей к данным, но при этом ограничивать объем данных, которые фактически могут просматривать пользователи. Если пользователям предоставляется доступ ко всему источнику данных, то они приобретают возможность просматривать все данные, доступные с помощью этого источника данных. Представления источников данных позволяют вместо всего множества данных источника задать лишь подмножество первоначального списка доступных объектов.

Чтобы ввести новое представление источника данных, щелкните правой кнопкой мыши на элементе Data Source Views и выберите команду Add New Data Source View (рис. 17.8).

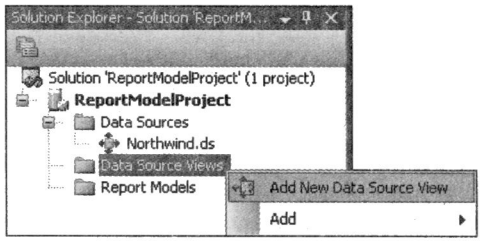


Рис. 17.8. Выбор команды Add New Data Source View

Появится очередное диалоговое окно приветствия "Welcome to the wizard". Это окно не имеет никакого функционального назначения, поэтому щелкните на кнопке Next. Откроется диалоговое окно, которое уже было описано выше и фактически идентичное показанному на рис. 17.5 (не считая одного-двух весьма незначительных отличий по сравнению с теми элементами, которые использовались перед этим для создания источника данных).

Подтвердите значения, заданные в этом окне по умолчанию, а затем несколько раз щелкните кнопкой мыши, чтобы перейти к диалоговому окну Select Tables and Views (рис. 17.9). Это окно, как свидетельствует его название, позволяет определить таблицы и представления, которые должны находиться в распоряжении конечных пользователей создаваемой модели отчетов.

Вначале выберите объект Order Details и воспользуйтесь кнопкой >, чтобы перенести его в столбец Selected Objects. Затем щелкните на кнопке Add Related Tables (Добавить соответствующие таблицы) и понаблюдайте над тем, что происходит, — в программном обеспечении SQL Server будут применены заданные правила о связях между таблицами (которые основаны на определениях внешних ключей) для получения информации о том, какие таблицы связаны с одной или несколькими таблицами, которые уже были выбраны. В данном случае с помощью программного обеспечения выбираются таблица Orders, родительская по отношению к таблице Order Details, и Products (поскольку в таблице Order Details имеется внешний ключ, указыва-

ющий также на эту таблицу). Дополнительно выберите таблицу `Customers`, чтобы завершить подбор таблиц (см. рис. 17.9).

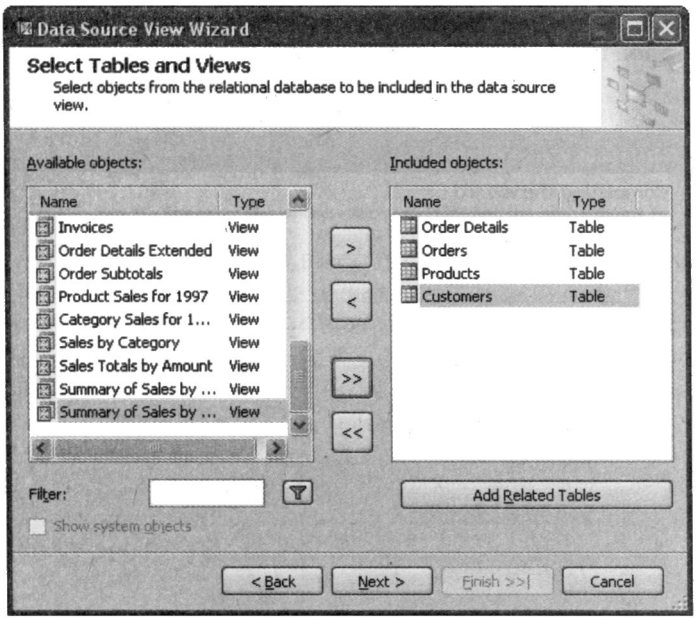


Рис. 17.9. Диалоговое окно Select Tables and Views

На данном этапе опции Filter и Show system objects в окне Select Tables and Views не представляют для нас интереса. Кратко можно отметить, что опция Filter позволяет определить условия включения объектов в список, чтобы было проще работать с базами данных, которые включают очень большое количество объектов, подлежащих отображению. Опция Show system objects имеет назначение, полностью соответствующее ее названию, – она позволяет включать модель отчетов в системные объекты (по мнению автора, в подавляющем большинстве приложений применение этой опции полностью лишено смысла).

Щелкните на кнопке Next, после чего наконец-то откроется диалоговое окно Completing the Wizard (рис. 17.10). Оно содержит краткую сводку требований к данным, которые должны быть включены в создаваемое представление источника данных. К тому же это окно позволяет проверить введенные данные, прежде чем завершить работу и подтвердить правильность ввода.

Затем щелкните на кнопке Finish, и представление источника данных будет фактически создано и добавлено к модели `ReportModelProject`.

Теперь мы можем приступить к созданию модели отчета. Как и при создании источника и представления данных, щелкните правой кнопкой мыши на узле Report Models дерева Solution Explorer и выберите команду Add New Report Model (рис. 17.11).

Прежде чем начнут открываться окна, позволяющие определить модель отчета, появится еще одно диалоговое окно с приветствием. Щелкните в этом окне на кнопке Next, после чего откроется более содержательное диалоговое окно Select Data Source View (Окно выбора представления источника данных), показанное на

рис. 17.12. Как и следовало ожидать, в этом окне показано единственное, только что созданное представление (которое также имеет имя `Northwind`).

Выберите в этом окне единственное имеющееся в нем представление источника данных и щелкните на кнопке Next, чтобы перейти к следующему диалоговому окну (рис. 17.13), которое позволяет определить правила формирования модели отчета.

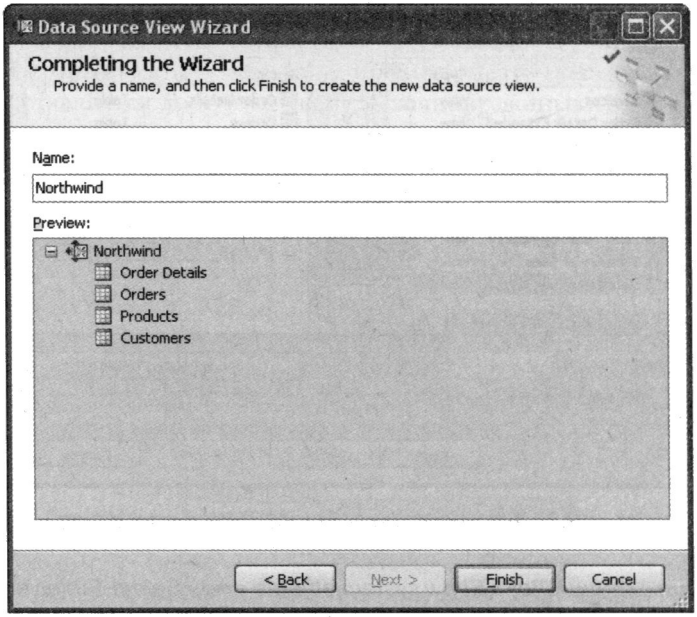


***Рис. 17.10. Диалоговое окно** Completing the Wizard*

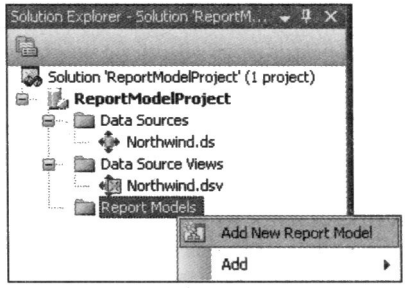


***Рис. 17.11. Выбор команды** Add New Report Model*

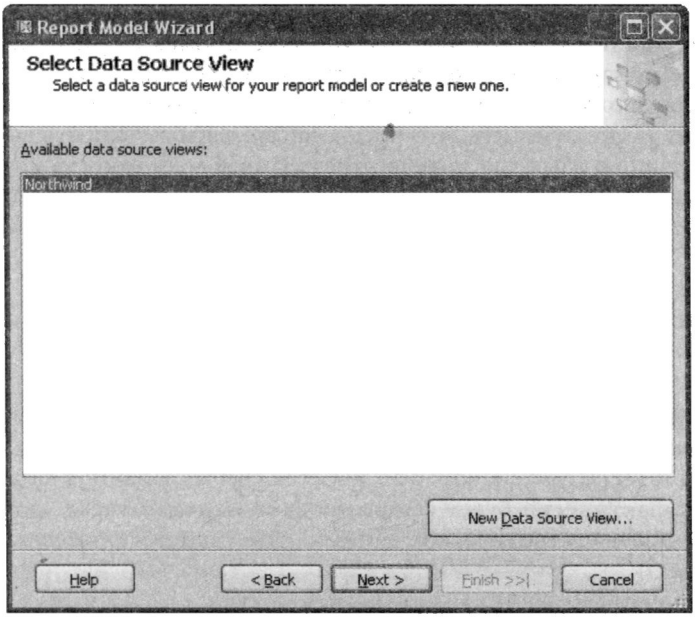


Рис. 17.12. Диалоговое окно Select Data Source View

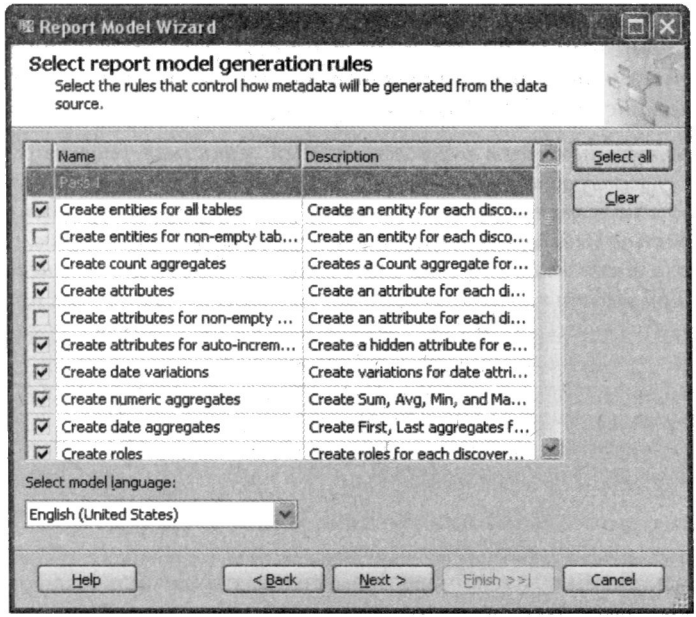


Рис. 17.13. Диалоговое окно Select report model generation rules

Правила формирования модели отчета позволяют определить такие характеристики отчета, которые показывают, например, какие промежуточные итоги позволяет формировать используемая модель отчета. Кроме того, эти правила содержат другие рекомендации, позволяющие упростить для конечных пользователей формирование отчетов. Следует также отметить, что предоставляется возможность выбирать применяемый по умолчанию язык для модели отчета (такая возможность является удобной для международных компаний или таких компаний, в которых приходится формировать отчеты на национальных языках, не предусмотренных в базовой инсталляции сервера).

Продолжите работу и подтвердите заданные в этом окне значения, предусмотренные по умолчанию, щелкнув на кнопке Next. В открывшемся диалоговом окне Update Statistics (Обновить статистические данные) можно выбрать один из вариантов обновления статистических данных.

- Update statistics before generating. Если выбрать этот вариант обновления статистических данных, то будут обновлены все статистические данные по всем таблицам и индексам, на которые имеется ссылка в данной модели отчета, до того, как фактически будет сформирована модель отчета (дополнительную информацию о статистических данных, относящихся к таблицам и индексам, см. в главе 9).
- Use current statistics in the data source view. Применение этого варианта равносильно указанию на то, что должны использоваться существующие статистические данные. Если количество используемых таблиц и индексов велико, то применение этого варианта позволяет сэкономить много времени при формировании отчета (поскольку не приходится ожидать обновления всей статистической информации), но при этом возникает риск получения модели отчета, созданной на основе предположений, не отражающих тот состав данных, который сложился к этому моменту.

Принципы, которыми следует руководствоваться, принимая решение о том, следует ли использовать существующие статистические данные или выполнять дополнительные операции по обновлению статистических данных, являются довольно сложными. По мнению автора, желательно выбирать в диалоговом окне Update Statistics *опцию, предусмотренную по умолчанию, и всегда обновлять статистические данные. Иное решение может быть принято только при условии полного понимания того, по каким причинам пропускается этап обновления статистических данных, с учетом всех последствий этого решения. Безусловно, в большинстве инсталляций отказ от обновления статистических данных перед созданием модели отчета не приводит к каким-либо существенным нарушениям в работе, но именно данная ситуация относится к той категории, в которой из-за любого неучтенного нюанса могут возникать существенные отклонения от нормы и приводить к отрицательным последствиям, поэтому лучше перестраховаться, чем потом сожалеть о допущенной ошибке.*

Следующим (и последним) диалоговым окном является окно Completing the Wizard. Присвойте имя модели отчета (автор выбрал имя `Report Model Northwind`) и щелкните на кнопке Run, после чего будет создана модель отчета (рис. 17.14).

Щелкните на кнопке Finish, чтобы ознакомиться с результатами проделанной работы, которые должны выглядеть примерно так, как показано на рис. 17.15.

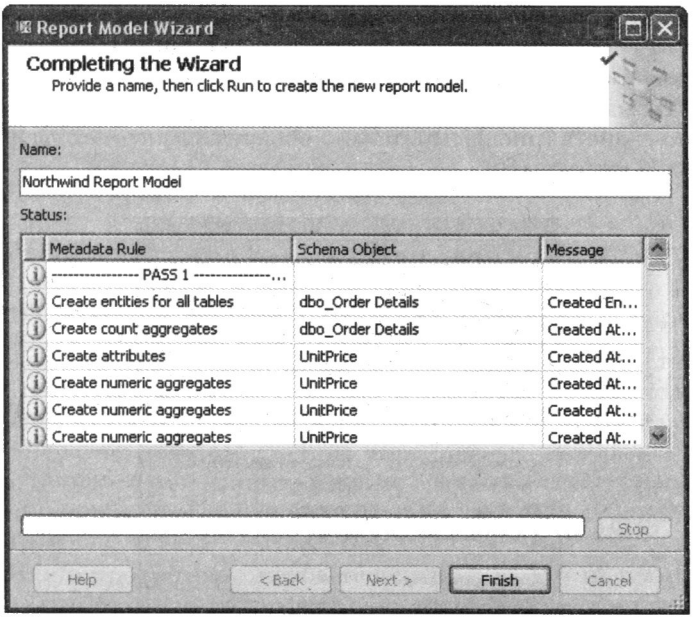


Рис. 17.14. Диалоговое окно Completing the Wizard

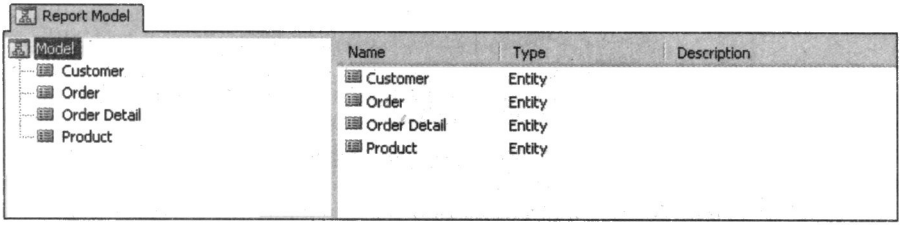


Рис. 17.15. Окно Report Model

Обязательно найдите время, чтобы ознакомиться с моделью отчета в окне Report Model. При составлении модели отчета в программном обеспечении SQL Server используются предположения в отношении того, что было и не было предусмотрено применять для вывода в виде отчета, но сделанный при этом выбор не всегда становится правильным (как будет вскоре продемонстрировано). Проверьте, что было включено в отчет, внесено дополнительно и сформировано по принципу создания производных атрибутов (в частности, рассмотрите промежуточные итоги и прочие элементы отчета).

К сожалению, для описания всех нюансов применения каждого элемента модели отчета и всех атрибутов этой модели не хватит целой главы. Мы можем лишь заострить внимание на том, что программное обеспечение SQL Server автоматически осуществляет разбивку полностью выясненных атрибутов (таких как даты) на меньшие атрибуты (скажем, отдельно указывая дни, месяцы и годы) с учетом общепринятых способов использования основополагающих типов данных этих атрибутов.

Как показывает рис. 17.15, автоматически сформированная модель отчета не включает все, что требуется, поэтому ее необходимо немного отредактировать.

Вначале перейдем к таблице `Order`. После щелчка на обозначении этой таблицы столбец `OrderID` выделяется серым цветом. Щелкнув на обозначении этого столбца и проверив окно свойств (рис. 17.16), можно обнаружить, что этот довольно важный столбец определен как скрытый.

Рис. 17.16. Окно Properties

Измените свойство `Hidden` так, чтобы оно имело значение `False`.

В данном случае мы сталкиваемся с одной из тех ситуаций, когда в проекте системы используется значение, автоматически сформированное с учетом того, какая информация должна предоставляться конечному пользователю. Например, значение в столбце идентификации вырабатывается системой, но фактически используется в качестве идентификатора заказа, `OrderID`, который видит заказчик в счете-фактуре. В столбцах идентификации часто применяются чисто служебные значения, не предназначенные для вывода во внешний интерфейс, именно поэтому в программном обеспечение SQL Server принято предположение, что значение идентификатора не должно появиться в создаваемом отчете.

В предыдущих разделах было кратко описано, как создать модель отчета, а в следующих разделах речь пойдет о том, как передать модель отчета в эксплуатацию.

Передача модели отчета в эксплуатацию

К счастью, задача ввода модели отчета в эксплуатацию является чрезвычайно простой. Наиболее сложная часть ее решения состоит в определении того, какое действие должно быть выполнено, а затем в составлении необходимой для этого команды. Чтобы ввести модель отчета в эксплуатацию, щелкните правой кнопкой мыши на свободном участке окна Solution Explorer (или на самой модели отчета) и выберите

команду Deploy. После этого остается только следить за тем, как происходит ввод модели отчета в эксплуатацию в окне вывода программы Visual Studio:

```
--- Deploy started: Project: ReportModelProject, Configuration: Production ---
Deploying to http://localhost/ReportServer?%2f
Deploying data source '/Data Sources/Northwind'.
Deploying model 'Northwind Report Model'.
Deploy complete -- 0 errors, 0 warnings
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
========== Deploy: 1 succeeded, 0 failed, 0 skipped ==========
```

Создание отчета

Сама модель отчета, безусловно, еще не отчет. В действительности модель отчета предназначена лишь для упрощения создания отчетов. Но этот подход, позволяющий упростить создание отчетов, является очень продуктивным, поскольку им могут пользоваться широкие круги пользователей из мира бизнеса, которые хорошо разбираются в отчетах, но не в базах данных. А после ввода в эксплуатацию модели отчета эти пользователи получают возможность создавать с помощью одной модели многочисленные отчеты.

Для выработки и просмотра отчетов необходимо выйти из программы Business Intelligence Studio и открыть интерфейс пользователя средств формирования отчетов, который по существу представляет собой Web-узел. Перейдите в браузере по адресу `http://<your reporting server host>/reports`. В случае, рассматриваемом автором, сервер развернут непосредственно в локальной системе, поэтому для перехода к интерфейсу формирования отчетов достаточно ввести в браузере адрес `http://localhost/reports` (рис. 17.17).

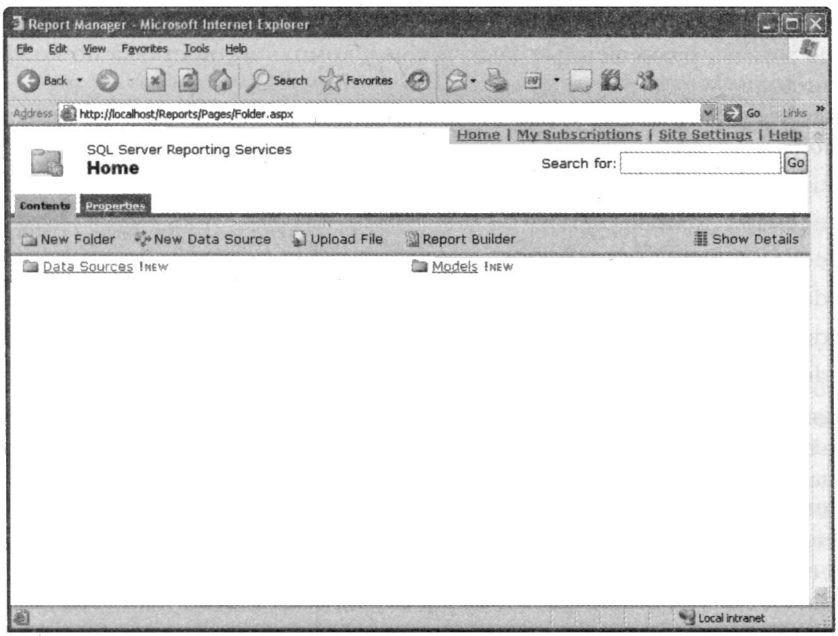


Рис. 17.17. Общий вид интерфейса формирования отчетов в окне браузера

По этому адресу находится начальная страница, позволяющая пользователям создавать отчеты и выполнять другие действия, предусмотренные в интерфейсе. Обратите внимание на то, что рядом с обозначениями источников данных (Data Sources) и моделей отчетов (Models) имеются надписи "!new" (этот замечательный факт свидетельствует о том, что наша попытка ввода в эксплуатацию модели отчета оказалась успешной!). Ознакомившись с этим интерфейсом, щелкните на ссылке Report Builder.

> Ссылка Report Builder указывает на небольшой аплет, попытка инсталляции которого в системе предпринимается после того, как пользователь в первый раз переходит по этой ссылке к программе формирования отчетов. Если пользователь намеревается в дальнейшем работать с программой Report Builder, то он должен подтвердить запрос на инсталляцию этого аплета.

Программа Report Builder выводит диалоговое окно с предложением выбрать источник данных для отчета. Выберите только что созданную модель отчета и щелкните на кнопке OK, чтобы получить применяемый по умолчанию шаблон отчета. Шаблон отчета представляет собой довольно удобную среду проектирования отчета, действующую по принципу переноса и фиксации, позволяющую просматривать таблицы, доступ к которым предоставлен с помощью модели отчета, и выбирать столбцы для последующего их перетаскивания в отчет. Обратите также внимание на крайнюю правую часть области окна, определяющей компоновку отчета. В ней предоставляется возможность выбора среди нескольких типичных компоновок, с которых можно начать формирование отчета.

В настоящем разделе будет рассматриваться отчет в табличной форме (такая компоновка является применяемой по умолчанию) и будет создан относительно простой отчет, содержащий информацию обо всех заказах, ожидающих обработки (таковыми считаются заказы, размещенные заказчиком в компании, поставка по которым еще не выполнена).

Прежде всего перейдите к каждой таблице и перетащите в отчет соответствующие столбцы:

- `Orders.OrderID`;
- `Customers.CompanyName`;
- `Products.ProductName`;
- `Order Details.Quantity`;
- `Orders.OrderDate`;
- `Orders.RequiredDate`.

Выполнив эту работу, щелкните в области заголовка и введите заголовок `Outstanding Orders`. После этого в окне появится примерно такое изображение, как показано на рис. 17.18.

Следует также отметить, что в окнах Report Data Explorer имена столбцов, введенных в отчет, обозначены полужирным шрифтом.

Закончив выполнение указанных действий, щелкните на элементе Run Report, после чего появятся первые плоды наших трудов (рис. 17.19).

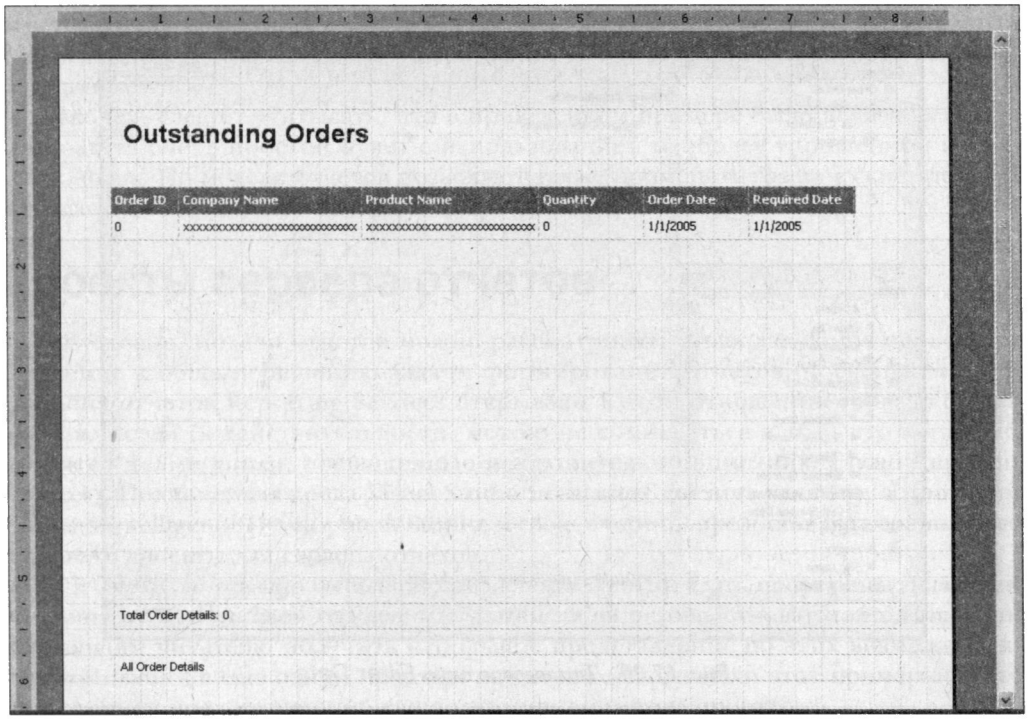


Рис. 17.18. Форма отчета Outstanding Orders

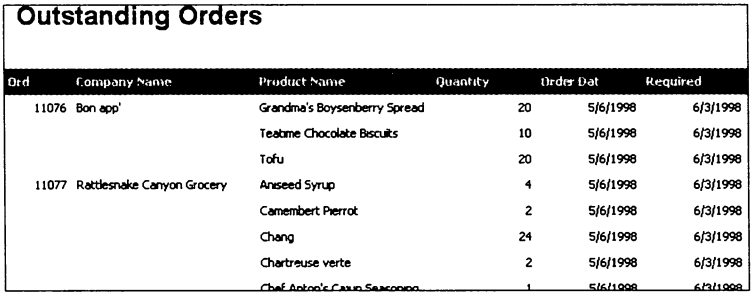


Рис. 17.19. Готовый отчет Outstanding Orders

Но тут возникает проблема, связанная с тем, что сформированный отчет состоит из 49 страниц. Безусловно, объем базы данных `Northwind` не мог самопроизвольно увеличиться до такой степени за то время, пока мы работали над этой книгой, поэтому следует вывод, что в отчет вошло слишком много данных. И действительно, в отчет включены все заказы, а не только заказы, оставшиеся необработанными. Однако программа Report Builder позволяет легко устранить эту ошибку. Достаточно щелкнуть на кнопке Filter панели инструментов и ввести с помощью диалогового окна Filter Data (рис. 17.20) ту информацию, которая в конечном итоге будет служить для создаваемого отчета в качестве конструкции `WHERE`.

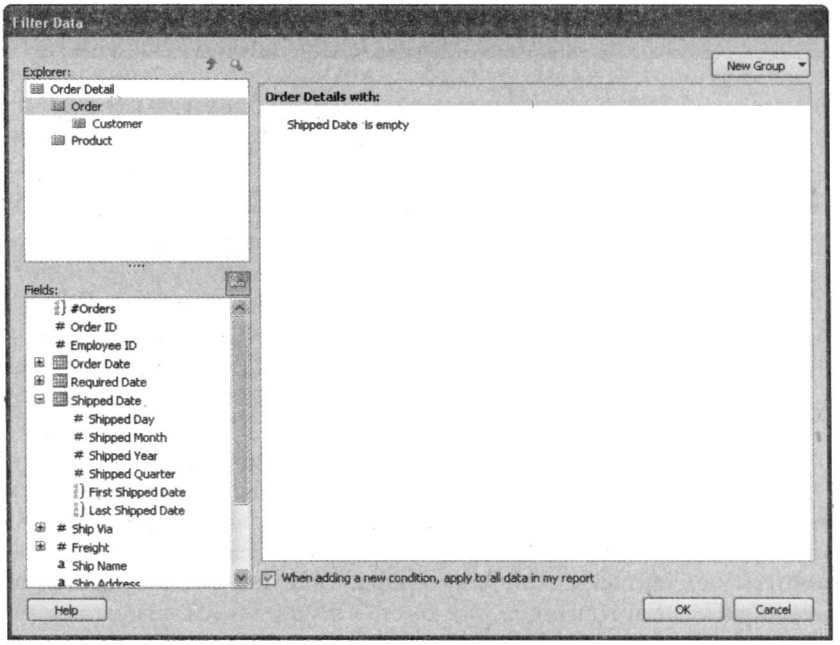


Рис. 17.20. Диалоговое окно Filter Data

В правую часть окна (см. рис. 17.20) автор перетащил обозначение столбца с датой поставки Shipped Date. Достаточно щелкнуть на применяемом по умолчанию обозначении операции сравнения "Equals" (равно) и выбрать вместо него вариант "Is Empty" (этот вариант эквивалентен конструкции `IS NULL` в стандартном языке SQL). Затем нужно щелкнуть на кнопке OK и еще раз вызвать отчет на выполнение, после чего количество страниц отчета станет примерно равным трем (поскольку общее количество строк в нем теперь составляет 73).

Проведите с интерфейсом формирования отчетов еще несколько экспериментов, и вы обнаружите, насколько легко можно добиться желаемых результатов с его помощью. Следует отметить, что можно было бы даже определить применяемый по умолчанию порядок сортировки, например, чтобы показать в первую очередь заказы, время выполнения которых подошло ближе всего. Для этого можно открыть диалоговое окно Sort and Group (рядом с окном Filter панели инструментов).

Некоторые дополнительные сведения о моделях отчетов

Безусловно, модели отчетов не позволяют удовлетворить все требования, а также осуществить все необходимые действия по формированию отчетов. Тем не менее они представляют собой великолепное новое средство SQL Server, позволяющее предоставить конечным пользователям доступ к данным и вместе с тем не терять контроля над действиями, которые выполняют пользователи (пользователи не могут просматривать больший объем информации по сравнению с тем, который был определен в источнике данных, а доступ к данным дополнительно защищен благодаря тому, что применение пользователями тех или иных источников данных строго контролируется). Вместе с тем пользователи могут создавать для себя именно такие

отчеты, которые их интересуют. Таким образом, модели отчетов предоставляют исключительно удобную возможность получения отчетов, которые требуются только на текущий момент.

Наконец, следует учитывать, что в приведенном примере было показано только, как создать очень простой отчет с использованием наиболее упрощенной из всех компоновок. Но модели отчетов позволяют также применять графические элементы и создавать отчеты более сложных, матричных форм.

Проекты сервера отчетов

Безусловно, модели отчетов можно рассматривать лишь как самое первое приближение к общему решению задачи формирования отчетов, поскольку службы создания отчетов Reporting Services открывают в этом отношении гораздо больше возможностей (в действительности, можно не сомневаться в том, что когда-либо появятся целые книги, посвященные исключительно описанию служб Reporting Services). Полноценная среда Visual Studio позволяет достичь многого, а программа Business Intelligence Development Studio, в свою очередь, предоставляет возможность создавать проекты для сервера отчетов.

Как уже было сказано, описанию этой темы должны быть посвящены целые книги, поэтому в настоящем разделе мы можем лишь руководствоваться подходом, позволяющим читателю получить небольшое представление об этих возможностях, ознакомившись с еще одним простым примером (фактически этот пример является повторением предыдущего, но с использованием метода проектов).

К этому времени читатель должен быть уже хорошо знаком со многими концепциями, на которых основан рассматриваемый пример, поэтому обойдемся без описания многочисленных копий экрана и приступим непосредственно к описанию нюансов, позволяющему сразу же получить полное представление о новых возможностях.

1. Откройте новый проект с помощью шаблона Report Server Project в программе Business Intelligence Development Studio.
2. Создайте новый источник данных, основанный на использовании базы данных `Northwind` (щелкните правой кнопкой мыши на обозначении каталога Shared Data Source и заполните открывшееся диалоговое окно; воспользуйтесь кнопкой Edit, если вам требуется вспомогательное диалоговое окно для подготовки строки соединения, или скопируйте строку соединения, полученную при выполнении предыдущих примеров в настоящей главе).
3. Щелкните правой кнопкой мыши на каталоге Reports и выберите команду Add New Report после щелчка в программе-мастере отчетов на почти уже раздражающем диалоговом окне Welcome.
4. Выберите источник данных, созданный в шаге 2, и щелкните на кнопке Next.

В результате должно появиться диалоговое окно проектирования запроса, Design the Query (рис. 17.21).

Безусловно, автор ввел текст запроса непосредственно; этот запрос примерно соответствует запросу, который использовался в отчете, созданном в разделе с описанием моделей отчетов, включая критерий отбора только тех заказов, по которым не выполнена доставка. Следует отметить, что на данном этапе вместо запроса можно было бы предусмотреть использование хранимой процедуры.

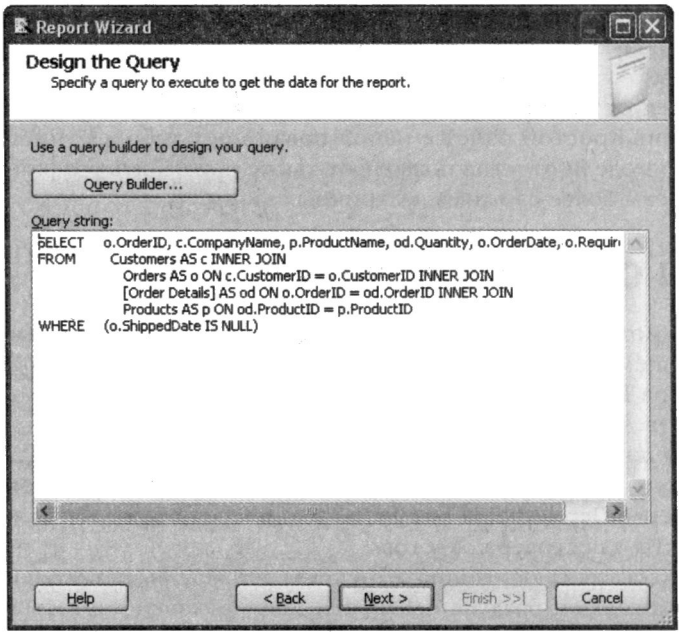


Рис. 17.21. Диалоговое окно Design the Query

Щелкните на кнопке Next и подтвердите применение отчета табличного типа, Tabular report. Откроется диалоговое окно Design the Table программы-мастера, показанное на рис. 17.22.

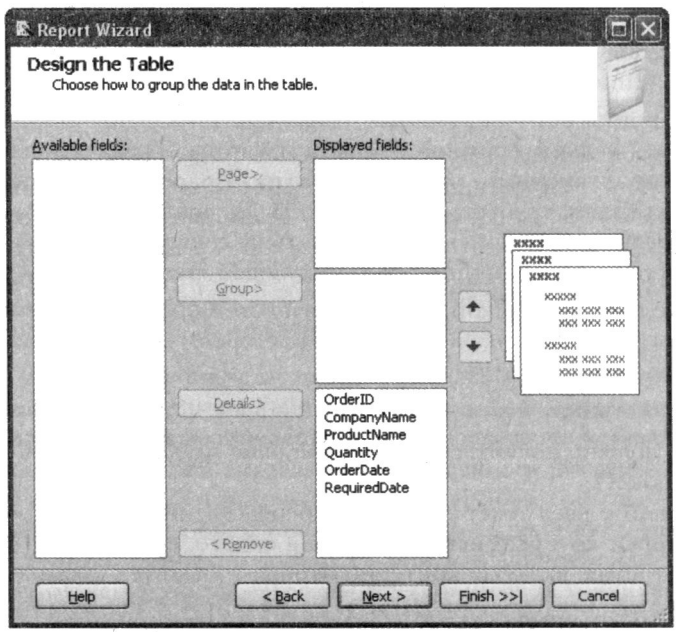


Рис. 17.22. Диалоговое окно Design the Table

Безусловно, в используемом запросе уже выбраны те столбцы, которые требуются для отчета (и, как оказалось, даже сами столбцы расположены в должном порядке, но при желании можно было бы откорректировать порядок расположения столбцов), поэтому в окне, показанном на рис. 17.22, были выбраны все столбцы и перенесены в поле Details.

На рис. 17.22 показаны также поля Page *и* Group, *которые позволяют задавать варианты выбора иерархии сортировки. Например, если бы потребовалось распределить данные об отдельных заказчиках по разным страницам (допустим, чтобы специалистам коммерческого отдела было проще понять, как складывается работа с конкретными заказчиками), то можно было бы перенести обозначение столбца* `CompanyName` *на уровень страницы* Page. *Аналогичным образом, вместо этого можно было бы применить группирование (а не разбивку на страницы) с учетом наименования товара, чтобы пользователи, исполняющие заказы путем списания товаров со складского учета, могли получить за один проход информацию обо всех товарах, необходимую для оформления всех заказов, ожидающих обработки.*

Снова щелкните на кнопке Next, чтобы открыть диалоговое окно Table Style. Выберите требуемый стиль оформления таблицы (сам автор неизменно использует стиль Bold) и еще раз щелкните на кнопке Next, чтобы открыть окно со сводкой заданных требований к отчету. Щелкните на кнопке Finish, чтобы создать определение отчета, как показано на рис. 17.23 (если был бы выбран другой стиль оформления, то определение отчета выглядело бы иначе).

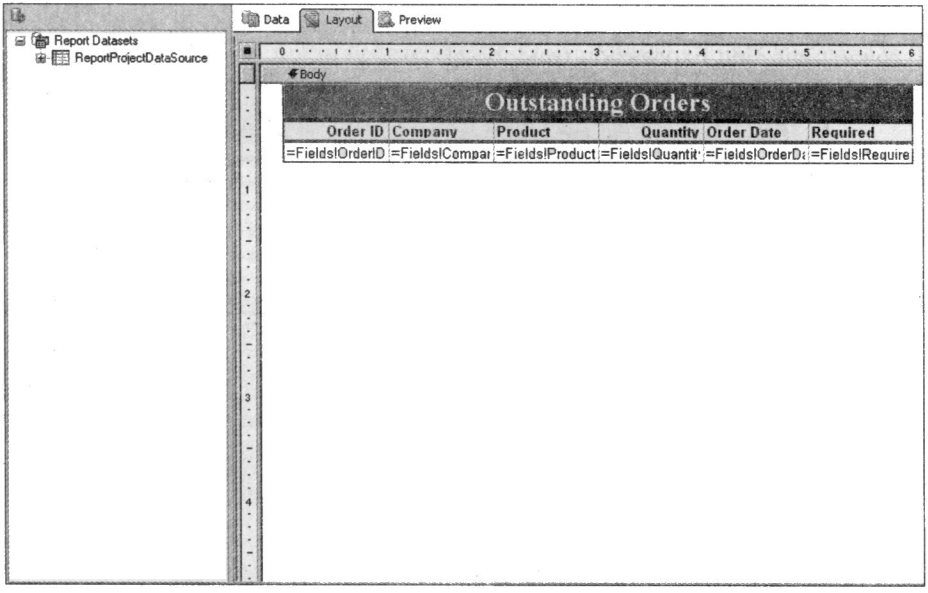


Рис. 17.23. Определение отчета Outstanding Orders

Читатели, знакомые с другими программами проектирования отчетов, обнаружат, что описанный здесь редактор с непосредственным отображением во многом напоминает другие известные программы, поскольку применяемые в них средства в основном стандартизированы (разумеется, между этими программами часто обнару-

живаются значительные различия, но способы представления данных главным образом остаются относительно неизменными).

После перехода к этапу предварительного просмотра отчета можно обнаружить, что полученный отчет почти не отличается от отчета, сформированного с помощью модели отчета. Однако нельзя не отметить наличие некоторых областей применения средств форматирования, в которых применяемые по умолчанию параметры программы разработки модели отчета, по-видимому, немного лучше тех, которые предоставляются в программе создания проекта отчета. В частности, отметим, что нет смысла включать в обозначение даты компонент, представляющий время, если значение времени всегда соответствует полуночи. С учетом этого внесем некоторые изменения в отчет, сформированный программой-мастером.

Практическое занятие

Внесение изменений в проект отчета

В предыдущем разделе были показаны действительно превосходные возможности создания отчетов с помощью программы-мастера буквально за считанные секунды. Однако полученный отчет оказался не идеальным, поэтому необходимо применить более удобный формат представления дат, как описано ниже.

1. Щелкните правой кнопкой мыши на первом поле с датой (Order Date) и выберите команду Properties.
2. Щелкните на обозначении fx для указания на то, что требуется изменить способ вывода так, чтобы он стал результатом вызова функции. Откроется диалоговое окно Edit Expression (рис. 17.24).

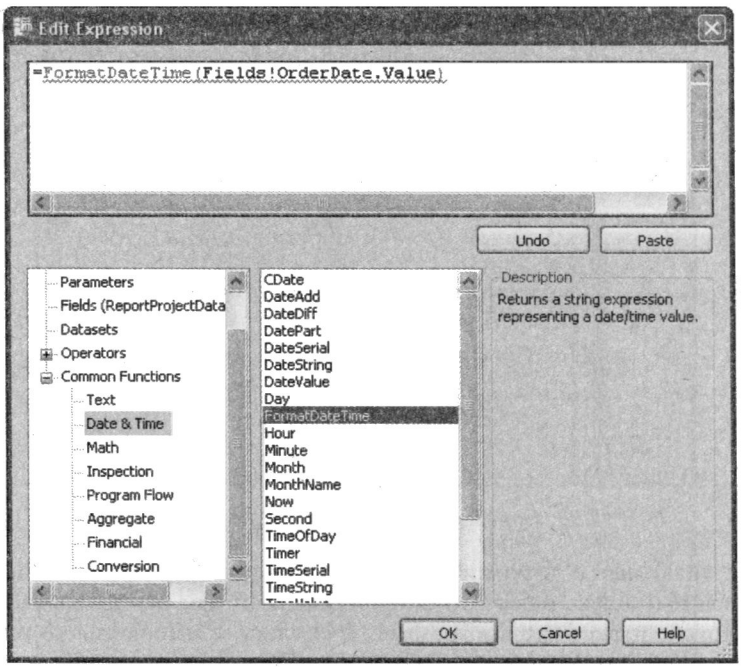
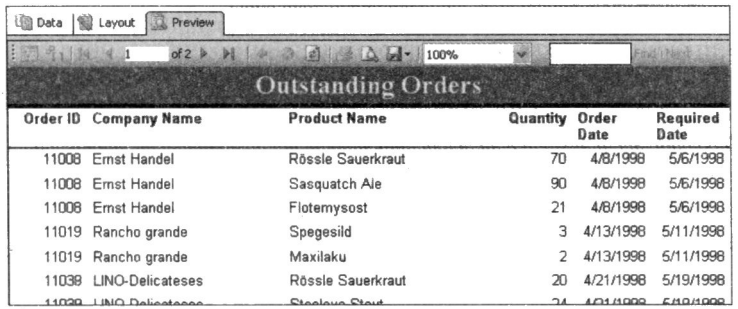


Рис. 17.24. Диалоговое окно Edit Expression

3. Укажите, что требуется функция `FormatDateTime`. Обратите внимание на то, что под кнопками, находящимися в верхней части, развернуто окно со справочной информацией о функциях. Теперь можно дважды щелкнуть на обозначении функции, и имя этой функции появится в верхней части окна. При вводе параметров функции предусмотрена возможность воспользоваться контекстно-зависимыми всплывающими подсказками, аналогичными тем, которые предоставляются в других частях программы Visual Studio. Следует также отметить, что данная конкретная функция имеет необязательный параметр, который позволяет задавать определенный стиль представления даты (допустим такой, который принято использовать в европейских странах или в Японии), но в данном случае решено, чтобы в функции использовался формат, соответствующий параметрам локализации, заданным на сервере.
4. Щелкните на кнопке OK и откройте вкладку Format. Установите значение опции Direction, равное RTL (Right To Left — справа налево), чтобы выравнивание данных в этом поле отчета осуществлялось по правому краю (такой способ выравнивания типичен для большинства отчетов).
5. Щелкните на кнопке OK и повторите тот же процесс для столбца `RequiredDate`.
6. Снова перейдите в режим предварительного просмотра отчета, выбрав вкладку Preview. Откроется окно, которое должно выглядеть примерно так, как показано на рис. 17.25.



Order ID	Company Name	Product Name	Quantity	Order Date	Required Date
11008	Ernst Handel	Rössle Sauerkraut	70	4/8/1998	5/6/1998
11008	Ernst Handel	Sasquatch Ale	90	4/8/1998	5/6/1998
11008	Ernst Handel	Flotemysost	21	4/8/1998	5/6/1998
11019	Rancho grande	Spegesild	3	4/13/1998	5/11/1998
11019	Rancho grande	Maxilaku	2	4/13/1998	5/11/1998
11038	LINO-Delicateses	Rössle Sauerkraut	20	4/21/1998	5/19/1998

Рис. 17.25. Внешний вид отчета на вкладке Preview

Следует отметить, что автор немного откорректировал значение ширины столбцов отчета (попытайтесь и вы это сделать!), чтобы пространство печатной страницы использовалось более эффективно.

Описание полученных результатов

При осуществлении описанных выше действий незаметно для пользователя применяется так называемый язык определения отчетов (Report Definition Language — RDL). Язык RDL фактически реализован на основе языка XML. При внесении любых изменений автоматически модифицируются определения RD (Report Definition), что позволяет во время генератора отчетов учитывать все заданные требования.

Чтобы ознакомиться с тем, как выглядит определение RDL, щелкните правой кнопкой мыши в окне Solution Explorer и выберите команду View Code. Объем полученной при этом информации очень велик, но ниже приведена небольшая выдержка из определения, касающаяся одного из отредактированных нами полей отчета.

```
<TableCell>
   <ReportItems>
   <Textbox Name="OrderDate">
      <rd:DefaultName>OrderDate</rd:DefaultName>
      <ZIndex>1</ZIndex>
      <Style>
      <PaddingLeft>2pt</PaddingLeft>
      <PaddingTop>2pt</PaddingTop>
      <PaddingBottom>2pt</PaddingBottom>
      <Direction>RTL</Direction>
      <PaddingRight>2pt</PaddingRight>
      </Style>
      <CanGrow>true</CanGrow>
      <Value>=FormatDateTime(Fields!OrderDate.Value)</Value>
   </Textbox>
   </ReportItems>
</TableCell>
```

После глубокого освоения всех необходимых для этого сведений разработчик получает возможность редактировать такие определения RDL вручную.

Ввод проекта отчета в эксплуатацию

После разработки проекта отчета можно приступить к вводу его в эксплуатацию. Как и при использовании подхода, основанного на модели отчета, для этого достаточно щелкнуть правой кнопкой мыши на обозначении отчета в окне Solution Explorer и выбрать команду Deploy. Тем не менее необходимо учитывать один небольшой, но важный нюанс — в определении проекта должен быть указан адресат развертывания. А после развертывания отчета по указанному адресу можно приступать к его эксплуатации.

1. Щелкните правой кнопкой мыши на названии вновь созданного проекта сервера отчетов и выберите команду Properties.
2. Откроется окно, в поле TargetServerURL которого необходимо ввести URL сервера отчета. На компьютере автора в качестве такого URL достаточно было указать `http://localhost/ReportServer`, но в качестве сервера может быть задан любой сервер, на котором вы имеете соответствующие права развертывания проектов отчетов (обозначение сервера отчетов как `ReportServer` также может быть другим, если в качестве значения параметра Virtual Directory во время инсталляции было указано иное имя).

После ввода проекта отчета в эксплуатацию можно приступить к просмотру отчета. Введите в браузере адрес сервера отчетов (если сервер отчетов эксплуатируется на локальном хосте и в нем используется каталог, предусмотренный по умолчанию, то для перехода к начальной странице сервера отчетов можно ввести адрес `http://localhost/Reports`, как и в приведенных выше примерах применения моделей отчетов). В открывшемся окне можно сразу же обнаружить, что к перечню ссылок добавлена ссылка на проект сервера отчетов. В данном случае был создан проект ReportProject (рис. 17.26).

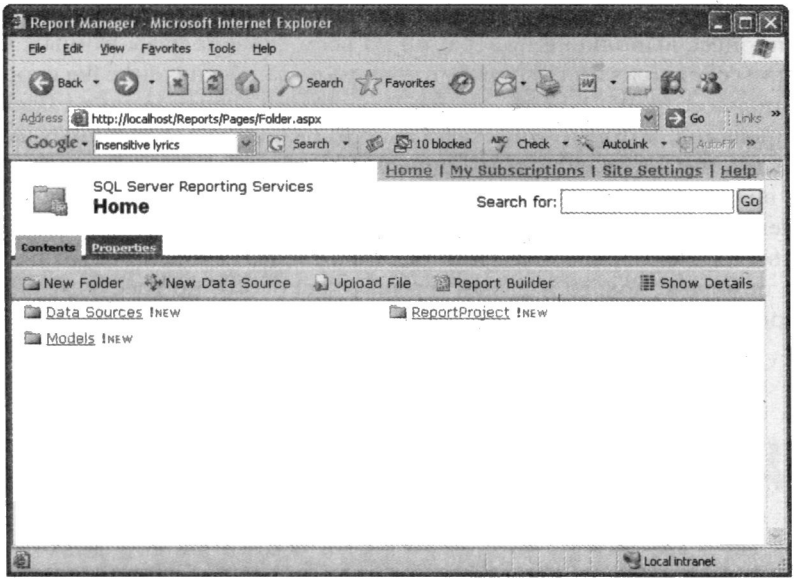


Рис. 17.26. Проект ReportProject

Щелкните на этой ссылке, которая указывает на проект отчета, и выберите отчет с данными о заказах, ожидающих обработки. Для создания отчета после первой его загрузки потребуется определенное время (а если вы снова перейдете к указанной ссылке, то обнаружите, что на этот раз отчет будет создан довольно быстро, поскольку его определение уже находится в кэше), но в конечном итоге сформируется отчет, который полностью соответствует определению, заданному в проекте отчета.

Резюме

Службы Reporting Services впервые появились в версии SQL Server 2005, поэтому еще рано судить о том, насколько широкое применение эти службы найдут на предприятиях, эксплуатирующих СУБД SQL Server, и в проектах, созданных на основе этой СУБД. Ясно одно — эти службы характеризуются наличием весьма значительного набора функциональных средств, а возможности, связанные с использованием проектов отчетов, являются буквально безграничными. Несмотря на это, опыт автора подсказывает, что отделы информационных технологий не будут поддерживать идею развертывания Web-сервера наряду с SQL-сервером (сервер отчетов не обязательно должен находиться на том же компьютере, что и сервер реляционной машины, но при использовании для их эксплуатации разных компьютеров процедура инсталляции становится немного сложнее). Кроме того, я пришел также к выводу, что эти отделы часто выступают против использования таких конфигураций, в которых допускается развертывание новых функциональных средств (в данном случае новых отчетов) не под их контролем. Безусловно, они имеют весьма основательные причины высказывать свои возражения против подобных вариантов "динамического" развертывания, поскольку даже единственный запрос, который плохо структурирован или

не имеет надлежащей поддержки в виде индексов, способен существенно замедлить обычный процесс выполнения операций на всем сервере базы данных. Отделы информационных технологий несут ответственность за обеспечение бесперебойной работы всех пользователей, поэтому вправе относиться скептически к таким средствам программирования, которые допускают использование непроверенных запросов.

Но в целом, несмотря на то, что многие отделы информационных технологий не торопятся внедрять службы Reporting Services, эти службы представляют собой вполне приемлемое дополнение к обычным средствам взаимодействия с пользователем, которое позволяет разработчикам передать конечным пользователям некоторые функции с помощью нового и разностороннего способа, не требуя от них освоения навыков программирования.

18

Обеспечение интеграции с помощью служб Integration Services

Настоящая глава полностью посвящена описанию новых средств преобразования данных СУБД SQL Server. Предварительно в ней дано краткое описание служб преобразования данных (Data Transformation Services – DTS), на которых основаны эти новые средства. Но на смену им пришли службы Integration Services, и данная глава посвящена их описанию.

Мы начинаем изложение сведений о современных средствах преобразования данных с описания DTS по двум причинам. Во-первых, ввод в действие этих служб был буквально революционным шагом. До сих пор никогда еще в одну из основных реляционных СУБД не включали столь значительный набор инструментальных средств перемещения и трансформации крупных блоков данных. Благодаря этому всевозможные сложные операции, которые до сих пор либо были очень трудоемкими, либо требовали применения весьма дорогостоящих инструментальных средств независимых разработчиков, внезапно удалось существенно упростить. Во-вторых, эти службы, ставшие основой современных служб, теперь исключены из программного обеспечения SQL Server или, вернее, заменены, поэтому их описание трудно найти в другом источнике.

Программное обеспечение служб DTS при подготовке последнего выпуска SQL Server было полностью модернизировано и получило новое название – Integration Services.

Если вы эксплуатируете комплексную среду, в которой применяются одновременно версии SQL Server 2000 и SQL Server 2005, или намереваетесь перейти со старой версии на новую, то будете иметь возможность эксплуатировать службы DTS без каких-либо проблем. Дело в том, что службы Integration Services версии SQL Server 2005 позволяют применять имеющиеся пакеты DTS после инсталляции служб Legacy Services с помощью программы-мастера Installation Wizard во время инсталляции SQL Server 2005.

А программа-мастер SSIS Package Migration Wizard может помочь обновить имеющиеся пакеты DTS.

В настоящей главе показано, как выполнить основные операции импорта и экспорта данных, а также кратко описаны некоторые другие операции, осуществимые с помощью таких инструментальных средств, как Integration Services.

Общая постановка задачи

Задачи, которые могут быть решены с помощью служб Integration Services, обнаруживаются в той или иной форме почти во всех производственных системах. Дело в том, что на практике часто возникает необходимость обеспечить обмен данными в прямом или обратном направлении между СУБД и внешними источниками данных. При этом приходится, в частности, импортировать данные из существующей системы в новую, обеспечивать передачу перечня имеющихся товаров от поставщика или выполнять тому подобные действия. Но все эти операции обмена данными характеризуются общей особенностью, заключающейся в том, что должен быть выполнен ввод или вывод данных в формате, который не совпадает с форматом данных в таблицах базы данных.

Потребность в осуществлении необходимых для этого действий была сформулирована как потребность в создании инструментального средства, позволяющего извлекать, преобразовывать и загружать данные в базу данных. Такое инструментальное средство обычно обозначают кратко с помощью аббревиатуры, состоящей из первых букв названий соответствующих операций, ETL (Extract, Transform, Load). Разработано много типов инструментальных средств вышеназванного назначения, которые обладают разными возможностями, но службы SQL Server Integration Services (или сокращенно SSIS) позволяют справиться почти с любой задачей, которая может встретиться на практике.

Программное обеспечение SSIS входит в состав версии SQL Server 2005, но не применяется для выполнения абсолютно всех операций обмена данными между СУБД SQL Server и внешними источниками данных. Причина этого заключается в том, что для эксплуатации в определенных вариантах межплатформенной среды предусмотрены некоторые другие программные пакеты независимых разработчиков, которые обеспечивают гораздо более удобное сопряжение разных платформ и имеют более привлекательный пользовательский интерфейс. Но в действительности эти программы рассчитаны на то, чтобы с их помощью неопытные пользователи могли относительно легко перемещать данные из одной системы в другую. Тем не менее они обладают также весьма существенным недостатком — являются чрезвычайно дорогостоящими. Даже в то время, когда еще был доступен лишь старый программный продукт, DTS, мне не раз приходилось наблюдать, как заказчики, эксплуатирую-

щие СУБД Oracle или другую СУБД, отличную от SQL Server, приобретали полную лицензию на SQL Server лишь для того, чтобы иметь возможность эксплуатировать службы DTS, и я уверен, что службы SSIS будут пользоваться не меньшим спросом (они являются исключительно удобными!).

Использование программы-мастера Import/Export Wizard для создания несложных пакетов

Пакет SSIS в службе SSIS по существу эквивалентен программе. Пакет объединяет в себе набор команд (возможно, включающий определенные ограниченные средства условного перехода), организованный так, что соответствующий фрагмент программного обеспечения можно передавать из одной точки в другую, распространять, редактировать и т.д. Программа-мастер Import/Export Wizard представляет собой инструментальное средство, позволяющее автоматизировать создание подобных пакетов, предназначенных для относительно простого осуществления операций импорта и экспорта.

Чтобы воспользоваться программой-мастером Import/Export Wizard, необходимо запустить инструментальное средство SQL Server Business Intelligence Suite из меню запуска программ операционной системы, а затем создать новый проект Integration Services.

Откровенно говоря, по мнению автора, перевод служб SSIS в состав инструментальных средств Intelligence Suite не вполне оправдан, хотя и следует знать, что инструменты, подобные SSIS, чаще всего применяются для извлечения данных из базы данных OLTP и преобразования этих данных для использования в базе данных OLAP. Однако службы SSIS могут применяться для осуществления гораздо более широкого спектра операций, поэтому, скорее всего, должны были быть включены в состав основных средств программы Management Studio (или по крайней мере доступ к этим службам нужно было предоставить с помощью обеих программ, Intelligence Suite и Management Studio).

Тем не менее еще раз отметим, что инструментальные средства SSIS входят в состав программного обеспечения Business Intelligence Suite, а не в программное обеспечение Management Studio, в отличие от многих других инструментальных средств, которые рассматривались в данной книге.

После того как будет выполнен запуск нового проекта Integration Services, в крайней правой части экрана в окне Solution Explorer найдите узел SSIS Packages, как показано на рис. 18.1 (окно Solution Explorer будет находиться в указанном месте, если вы еще не переместили ни одно из присоединяемых окон). Щелкните правой кнопкой мыши на обозначении SSIS Packages и выберите во всплывающем меню элемент SSIS Import and Export Wizard.

После этого откроется начальное диалоговое окно, которое обычно не используется, поэтому щелкните на кнопке Next, чтобы открыть гораздо более важное диалоговое окно Choose a Data Source, позволяющее выполнить настройку соединения с источником данных (рис. 18.2).

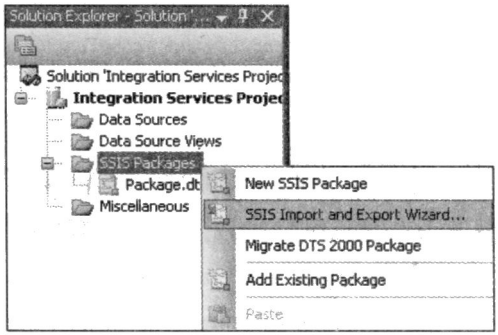


Рис. 18.1. Выбор команды SSIS Import and Export Wizard

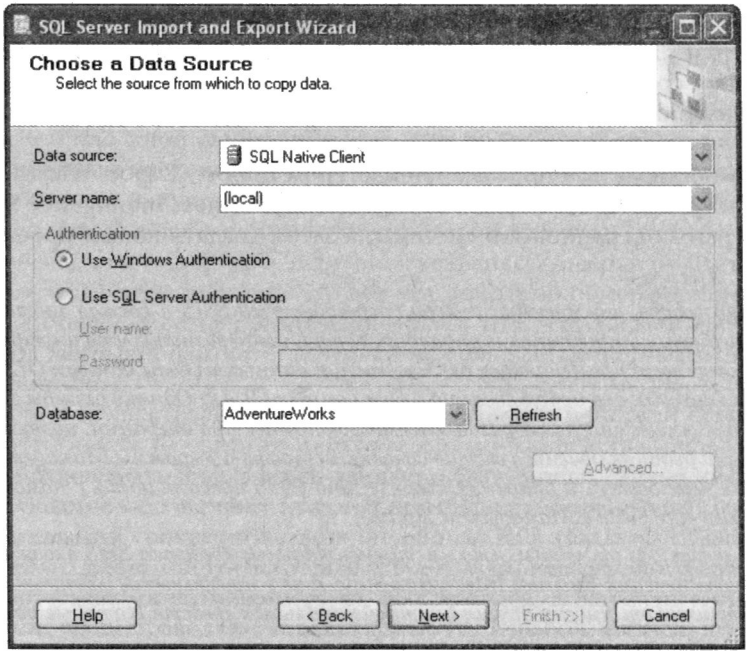


Рис. 18.2. Диалоговое окно Choose a Data Source

Корпорацией Microsoft было принято решение использовать в этом диалоговом окне одно и то же поле, Data Source, *для двух разных целей. В первом случае назначение этого поля соответствует всеобъемлющему определению источника данных, которое сопровождается всей информацией, необходимой для создания строки соединения (имя сервера, данные аутентификации, имя базы данных и т.п.). Во втором варианте применения введенная информация представляет собой субэлемент первого определения — обозначение источника данных как драйвера типа OLEDB.*

Не изменяйте пока фактически заданное содержимое поля со списком Data Source, т.е. обозначение SQL Native Client. Затем определите информацию аутентификации (сам автор в этом примере решил применить опцию Windows Authentication) и выберите в поле Database значение `AdventureWorks`.

После этого щелкните на кнопке Next, чтобы открыть во многом подобное диалоговое окно Choose a Destination. Но на этот раз в содержимое полей потребуется внести немного больше изменений. Измените значение поля Destination так, чтобы в нем появилось обозначение Flat File Destination (рис. 18.3).

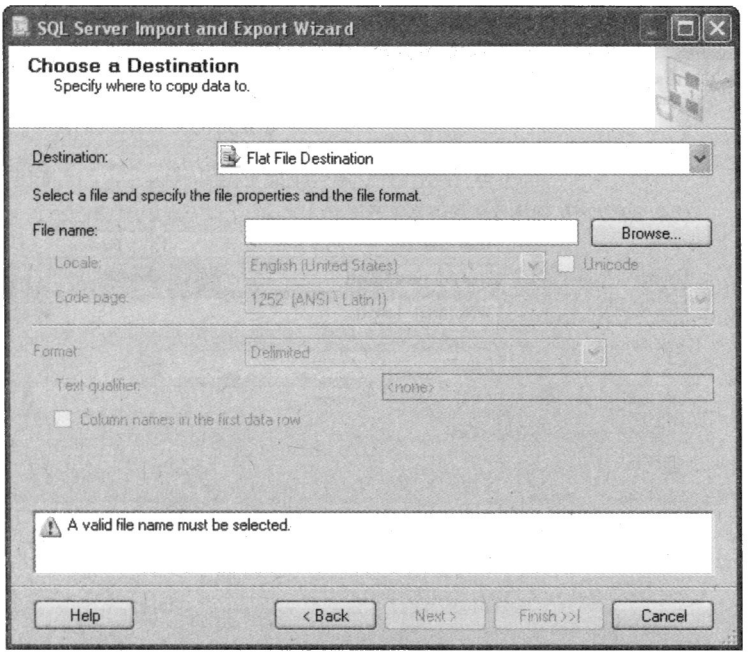


Рис. 18.3. Диалоговое окно Choose a Destination

Обратите внимание на то, что остальные поля изменятся в соответствии с тем, что является более подходящим для работы с файловой системой. Например, появится возможность задать имя файла в каталоге, заданном по умолчанию, вместо имени таблицы в базе данных SQL Server. А после щелчка на кнопке Browse появится стандартное диалоговое окно выбора файла. Присвойте файлу имя `TextExport.csv` и щелкните на кнопке OK. После этого для редактирования станут доступными некоторые другие опции (рис. 18.4).

В настоящем разделе не приводятся действительные сведения о выборе требуемых опций, поскольку состав этих опций в значительной степени зависит от того, какой именно источник данных OLEDB будет использоваться. Но в действительности вся суть применения этого диалогового окна на данном этапе уже становится очевидной. Программное обеспечение SSIS модифицирует диалоговое окно, предоставляя возможность выбирать опции, соответствующие характеристикам конкретного используемого источника данных OLEDB. В частности, при определении имен столбцов с данными, подлежащими выгрузке в текстовый файл, достаточно выбрать опцию Column names in first data row и щелкнуть на кнопке Next. Кроме того, для определения способа выборки данных из исходной базы данных можно использовать два варианта, описанных ниже.

- Copy. Этот вариант позволяет непосредственно скопировать таблицу или представление.
- Query. С помощью этого варианта может быть выбрана практически любая операция, позволяющая сформировать требуемый результирующий набор.

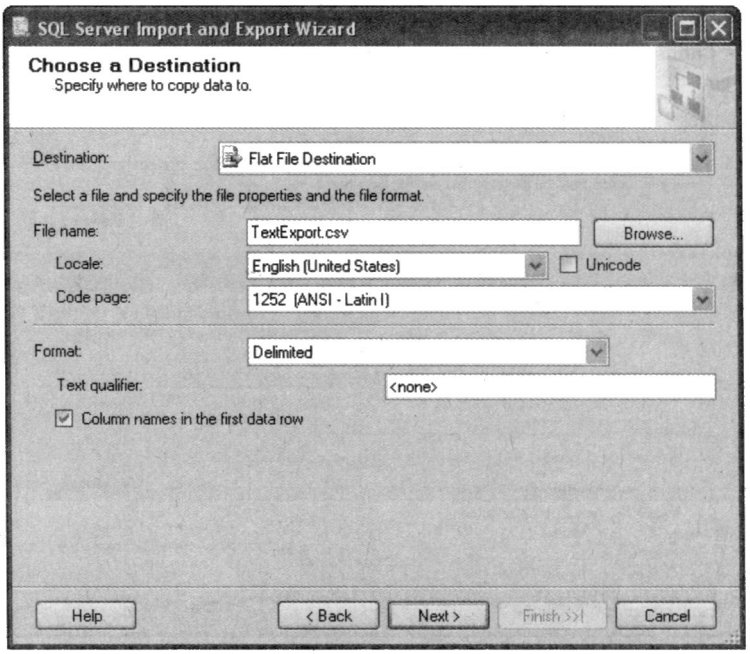


Рис. 18.4. Опции, доступные в диалоговом окне Choose a Destination

Вначале рассмотрим вариант Query. После того как вы выберете этот вариант и щелкнете на кнопке Next, откроется еще одно диалоговое окно, Provide a Source Query, применение которого потребует немного больше усилий по сравнению с предыдущим, поскольку в нем открывается область, в которой необходимо ввести текст запроса (рис. 18.5).

В приведенном на рис. 18.5 примере запроса показано, что есть возможность, например, экспортировать в плоский файл список служащих, которые были приняты на работу в первый месяц после открытия рассматриваемого делового предприятия. Чтобы проверить, какой результат формируется при выполнении данного примера, введите запрос, а затем проконтролируйте его, щелкнув на кнопке Parse, чтобы убедиться в том, что текст запроса, предназначенного для прогона, является синтаксически правильным. Затем щелкните на кнопке Next. Откроется диалоговое окно Configure Flat File Destination, показанное на рис. 18.6.

В окне, приведенном на рис. 18.6, определяются требования к внешнему виду создаваемого плоского файла. Чтобы убедиться в том, что введенный запрос возвращает именно те данные, которые соответствуют ожиданиям, щелкните на кнопке Preview (рис. 18.7).

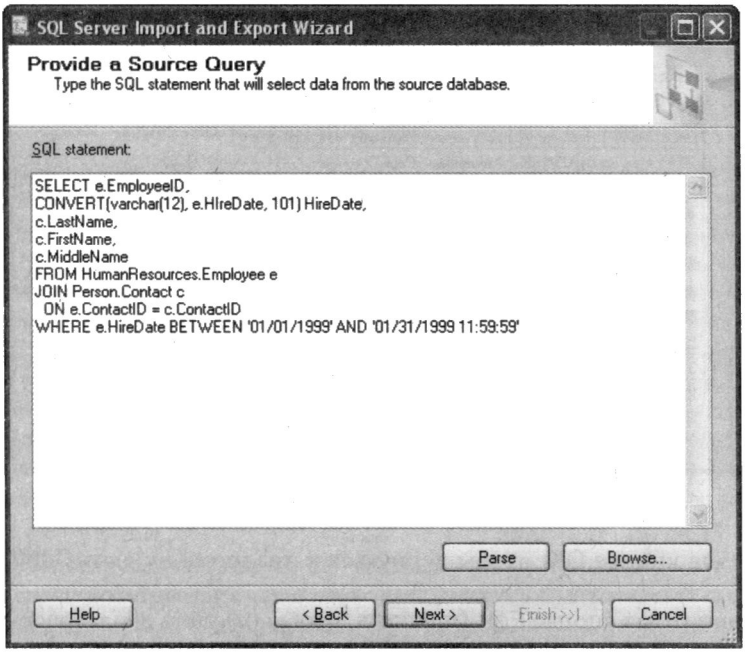


Рис. 18.5. Диалоговое окно Provide a Source Query

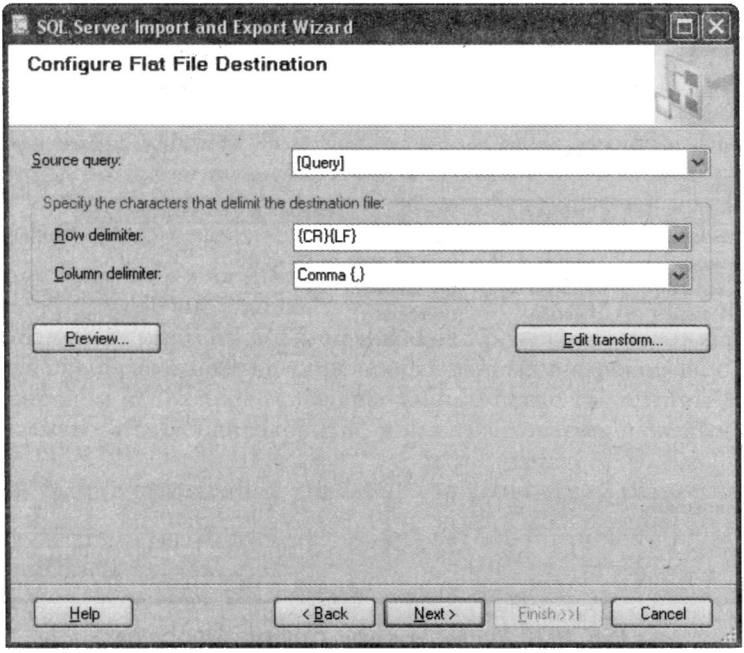


Рис. 18.6. Диалоговое окно Configure Flat File Destination

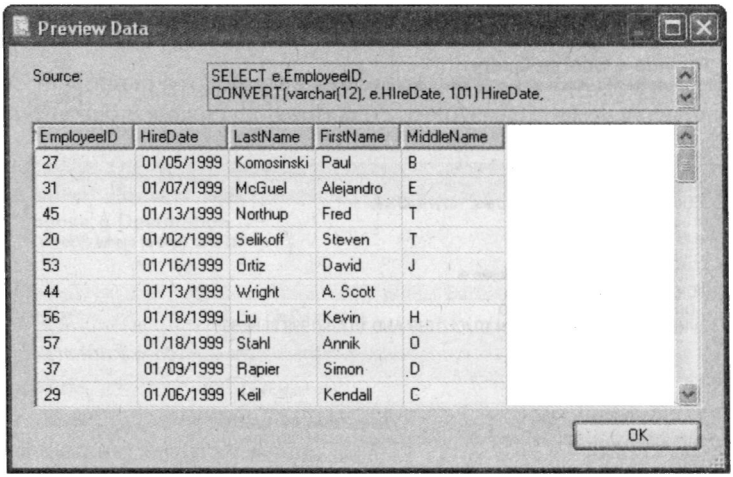


Рис. 18.7. Диалоговое окно Preview Data

Щелкните на кнопке OK, чтобы вернуться к диалоговому окну Configure Flat File Destination.

Затем щелкните на кнопке Edit Transform, чтобы открыть диалоговое окно Column Mappings (рис. 18.8).

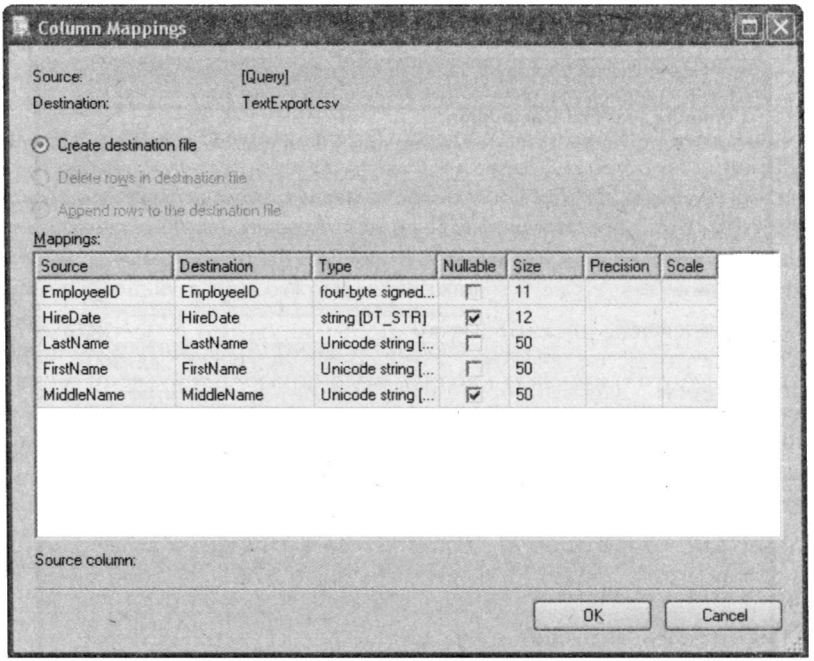


Рис. 18.8. Диалоговое окно Column Mappings

На первый взгляд может показаться, что окно, приведенное на рис. 18.8, не представляет особого интереса, но с помощью этого окна могут быть выполнены важные действия.

Прежде всего щелкните на одном из значений столбца Destination. Раскроется список опций, в том числе опция, позволяющая игнорировать данный столбец, а это, по существу, сводится к удалению исходного столбца из результирующего набора данных при выводе файла.

В связи с этим может возникнуть вопрос о том, для чего нужно предусматривать выборку данных столбца в первоначальном запросе, если эти значения должны игнорироваться в окончательном выводе. Но на это есть целый ряд причин. Прежде всего, может потребоваться получить данные столбца в основном для предварительного просмотра, чтобы убедиться в получении именно тех данных, которые следовало ожидать. Например, даже если требуется осуществить выборку только идентификаторов `EmployeeID`*, можно также включить столбец с данными о фамилиях, чтобы убедиться в том, что действительно осуществляется выборка данных о тех служащих, которые требуются. Кроме того, может оказаться, что используется запрос, который скопирован из какого-то другого приложения и не следует брать на себя риск его редактирования (например, если этот запрос является чрезвычайно сложным).*

Необходимо сделать еще одно замечание — диалоговое окно, показанное на рис. 18.8, формируется и в случае использования варианта с непосредственным копированием таблицы (напомним, что в одном из предыдущих окон, наряду с вариантом Query*, предусмотрен и вариант* Copy*), поэтому возможность исключения столбца с исходными данными из выходного набора данных становится еще более востребованной, поскольку в таблице может оказаться еще больше столбцов, которые не должны отображаться в окончательном выводе.*

Затем щелкните на одном из значений в столбце Type, после чего также откроется целый ряд вариантов преобразования типов данных.

Автор настоятельно рекомендует выбрать более подходящие значения ширины столбцов в диалоговом окне, показанном на рис. 18.8, чтобы было проще рассмотреть, какие значения приведены в столбце Type*. Для этого достаточно перетащить курсор мыши над правым краем заголовка столбца во многом по такому же принципу, как при использовании программы Excel, а затем щелкнуть и перетащить разделитель столбцов вправо, чтобы увеличить ширину столбца.*

В большинстве случаев достаточно остановиться на том способе преобразования типа, который выбран по умолчанию с учетом исходного типа данных, но иногда возникает необходимость каким-то образом модифицировать тип результата, например, предусмотреть преобразование целочисленных данных в строковые, с учетом того, для чего в конечном итоге предназначены данные (даже такие типы данных, которые внешне кажутся почти одинаковыми, могут трактоваться в разных системах поразному).

Наконец, в окне, показанном на рис. 18.8, предусмотрена возможность модифицировать значения полей Nullable, Scale и Precision. Необходимость в модификации этих значений возникает довольно редко, кроме того, сами принципы, которыми приходится руководствоваться при внесении соответствующих изменений, являются довольно сложными, но достаточно отметить, что при использовании более усовершенствованных способов преобразования (вместо предусмотренных по умолчанию в программе-мастере Import/Export Wizard) модификация значений указанных по-

лей могла бы стать основой перспективного способа обеспечения перехвата ошибок в данных, которые не удается успешно передать в систему, для которой предназначены данные. Тем не менее в большинстве случаев вместо подобной модификации указанных значений следует применять условия конструкции WHERE первоначального запроса.

Теперь щелкните на кнопке Cancel, чтобы вернуться к диалоговому окну Configure Flat File Destination, а затем на кнопке Next, чтобы перейти к следующему диалоговому окну.

На данном этапе откроется простое диалоговое окно, позволяющее подтвердить введенные параметры. В нем содержится краткая сводка всех требований, предъявленных программе-мастеру. Эти требования перечислены ниже.

- Скопировать строки из результирующего набора, созданного с помощью запроса, который определен в опции [Query], в файл C:\Documents and Settings\xxx\My Documents\TestExport.txt.
- Создать файл, в который должны быть выведены данные.
- Сохранить пакет в файле C:\Documents and Settings\xxxx\My Documents\Visual Studio 2005\Projects\Integration Services Project1\Integration Services Project1\Package1.dtsx.
- Не осуществлять немедленно прогон данного пакета.

Основная часть этих требований не нуждается в пояснениях, но необходимо подчеркнуть некоторые особенности.

Прежде всего, выходной файл с указанным именем не существует и должен быть создан. Если бы в ходе проектирования этого пакета было обнаружено, что выходной файл уже существует, то был бы предложен выбор — перезаписать существующий файл или добавить формируемые данные в его конец.

Но еще более важное замечание состоит в том, что создается и файл пакета. Пакет можно рассматривать как программу SSIS. Службы SSIS позволяют вызывать на выполнение один и тот же пакет снова и снова (в том числе автоматизировать его вызов с помощью расписания), и каждый раз с помощью этого пакета будет осуществляться определенная в нем операция экспорта.

Наконец, важно то, что, согласно предъявленным требованиям, прогон пакета не должен быть выполнен немедленно, поэтому необходимо либо вызывать его на выполнение вручную, либо запланировать его запуск.

Щелкните на кнопке Next, и пакет будет создан.

Необходимо еще раз подчеркнуть, что это последнее действие также относится к стадии подготовки — после создания пакет не запускается и данные не экспортируются. Чтобы действительно получить файл с экспортированными данными, все еще требуется вызвать пакет на выполнение или запланировать его выполнение. Но такая организация работы, связанная с запуском пакетов на выполнение, остается одинаковой применительно ко всем пакетам SSIS и не относится только к использованию программы-мастера Import/Export Wizard, поэтому мы также будем придерживаться этой организации работы до следующего раздела и перейдем к очень краткому описанию того, как обеспечить передачу данных в противоположном направлении, т.е. как импортировать данные.

Вызов пакетов на выполнение

Для вызова пакета SSIS на выполнение предусмотрено несколько различных способов, описанных ниже.

- Использование программы Execute Package Utility. По существу программа Execute Package Utility представляет собой исполняемый файл, с помощью которого можно определить пакет, который должен быть вызван на выполнение, задать все необходимые параметры и обеспечить запуск пакета с помощью этой утилиты по требованию.
- Вызов в качестве запланированной задачи с помощью программы SQL Server Agent. Дополнительные сведения о программе SQL Server Agent приведены в следующей главе, но на данный момент достаточно отметить, что вызов на выполнение пакета SSIS — это одно из заданий многих типов, для осуществления которых может применяться этот агент. Достаточно указать имя пакета, а также определить время и кратность его выполнения, после чего программа SQL Server Agent обеспечит требуемый запуск пакета.
- Вызов пакета из программы. В СУБД SQL Server предусмотрена целая объектная модель, обеспечивающая выполнение таких операций, как создание в программе экземпляров объектов SSIS, задание свойств для соответствующих пакетов и вызов их на выполнение. Тематика, связанная с этой объектной моделью, очень обширна, поэтому издательством Wrox выпущена целая книга, посвященная ее описанию (Knight et. al. *Professional SQL Server 2005 Integration Services*, Wiley, 2006). В связи с этим автор считает, что изложение данной темы выходит за рамки настоящей книги, и может лишь отметить, что к изучению этого предмета можно переходить только после соответствующей подготовки.

Использование программы Execute Package Utility

Программа Execute Package Utility представляет собой небольшой исполняемый файл, имеющий имя `DTExecUI.exe`. Вызов этой программы может осуществляться по такому принципу, что в строке вызова определяются параметры и значения, относящиеся к существующему пакету, после чего производится запуск пакета на выполнение. Предусмотрена также возможность открыть окно Windows Explorer, найти в файловой системе пакет (имена пакетов оканчиваются расширением `.DTSX`), а затем дважды щелкнуть на обозначении пакета, чтобы вызвать его на выполнение. Выполните указанные действия применительно к пакету, созданному по условиям описанного выше примера, чтобы открыть диалоговое окно Execute Package Utility (рис. 18.9).

Вполне очевидно, что с помощью этого окна можно открыть целый ряд различных диалоговых окон, которые можно выбирать, щелкая на названиях окон, приведенных в левой части указанного окна. Чтобы описать все варианты применения этих окон, потребовалась бы целая книга, поэтому в данном разделе описано лишь несколько наиболее важных особенностей нескольких основных диалоговых окон, предоставляемых в этой утилите.

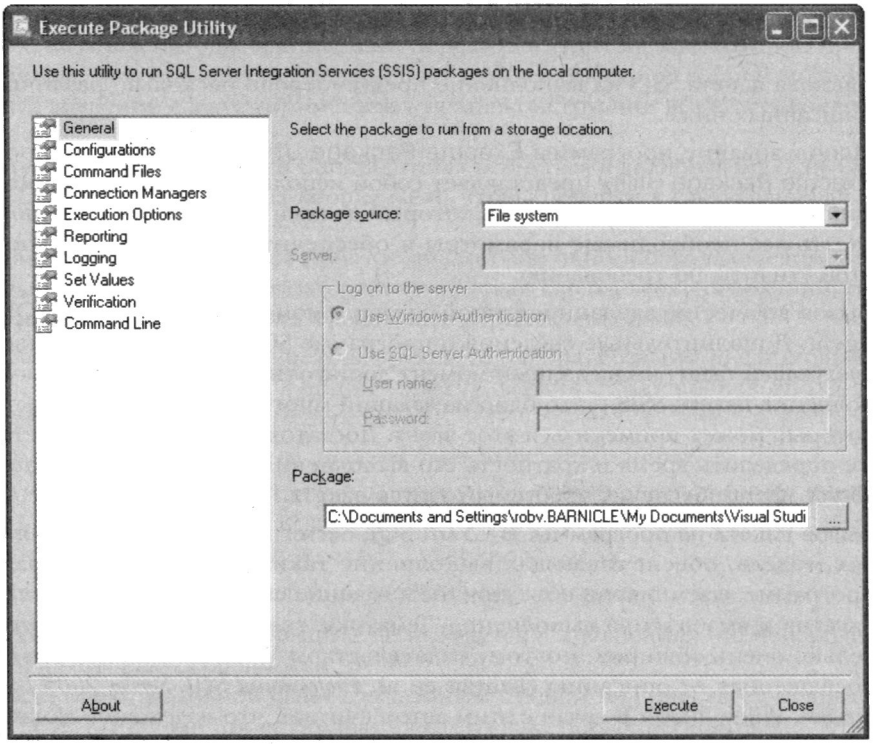


Рис. 18.9. Диалоговое окно Execute Package Utility

Диалоговое окно General

Основная часть полей в этом первом диалоговом окне, General, фактически не требует пояснений, но следует обратить особое внимание на содержимое поля Package Source. В соответствии со значением этого поля предусмотрена возможность применять для хранения пакетов SSIS любой из трех вариантов, описанных ниже.

- File system. Пример использования способа хранения пакетов File system (Файловая система) был приведен выше в описании программы-мастера Import/Export Wizard. Данный вариант является особенно удобным, если необходимо обеспечить переносимость пакетов, поскольку он позволяет легко сохранить пакет и перенести его в другую систему.
- SQL Server. Вариант SQL Server предусматривает хранение пакетов в СУБД SQL Server. При использовании такого подхода создается резервная копия пакетов во время каждого резервного копирования базы данных `MSDB` (это — системная база данных в каждой инсталляции SQL Server).
- SSIS Package Store. Модель хранения SSIS Package Store организована по принципу использования упорядоченного набора папок, в которых разрешается хранить определенные пакеты вместе с другими пакетами того же общего типа или назначения. Сами папки могут храниться либо в базе данных MSDB, либо в файловой системе.

Диалоговое окно Configurations

Программное обеспечение SSIS позволяет определять конфигурации для пакетов. Такие конфигурации представляют собой коллекции параметров, которые могут использоваться во время настройки конфигурации, а диалоговое окно Configurations позволяет комбинировать такие коллекции, создавая наборы параметров настройки.

Диалоговое окно Command Files

Диалоговое окно Command Files позволяет создавать командные файлы, которые представляют собой пакетные файлы, вызываемые на выполнение в составе пакета. Командные файлы могут использоваться для выполнения на уровне системы таких операций, как копирование файлов из одного каталога в другой, и других действий (для прогона этих командных файлов применяется учетная запись, в которой эксплуатируется служба Integration Services, поэтому указанной учетной записи должны быть предоставлены все необходимые права доступа к системе и сети).

Диалоговое окно Connection Managers

Название диалогового окна Connection Managers (Диспетчеры соединений) не совсем соответствует его назначению, поскольку в этом окне ведется не список диспетчеров соединений, а список соединений. Столбец Description, представленный в этом окне, позволяет ознакомиться со многими ключевыми свойствами каждого соединения, используемого в пакете. Следует отметить, что в описанном выше примере пакета применялись два соединения, причем более внимательный анализ показывает, что информация об одном из них относится к файлу (поскольку оно представляет собой соединение с используемым плоским файлом), а второе — непосредственно к СУБД SQL Server (это — соединение с источником экспорта).

Диалоговое окно Execution Options

Диалоговое окно Execution Options позволяет задавать опции этапа прогона. Не следует недооценивать важность этих опций. Они не только дают возможность указывать, какие в целом должны осуществляться действия, если возникнет какое-то нарушение в работе (например, из-за ошибки), но и позволяют отслеживать контрольные точки. Последняя возможность позволяет организовать контроль над тем, когда и где в пакете осуществляются различные этапы выполнения, что может оказаться очень важным с точки зрения настройки производительности и отладки.

Диалоговое окно Reporting

Диалоговое окно Reporting предназначено для получения максимального объема информации о происходящем. С помощью этого диалогового окна может обеспечиваться контроль над событиями, происходящими во время выполнения пакета. Объем полученной при этом информации зависит от того, какие события решено отслеживать и насколько подробной должна быть предоставляемая информация.

Диалоговое окно Logging

Задача настройки и эксплуатации средств контроля, предоставляемых диалоговым окном Logging, является весьма сложной, но ее решение позволяет получить очень большую отдачу, поскольку в свое распоряжение разработчик получает чрезвычайно

широкий набор средств, позволяющий контролировать работу даже наиболее сложных пакетов.

В целом диалоговое окно Logging позволяет настроить конфигурацию пакета для обеспечения записи информации в журнал с помощью целого ряда заранее настроенных программных средств, или так называемых провайдеров (по существу, провайдеры представляют собой четко определенные места назначения данных журнала). К числу заранее установленных провайдеров относятся текстовые файлы и даже таблицы SQL Server, но предусмотрена также возможность создавать собственные определяемые пользователем провайдеры (хотя эта задача является чрезвычайно сложной). Один из вариантов ведения журнала предусматривает запись информации на уровне пакета, а другой позволяет перейти к использованию весьма подробных степеней детализации и вести запись информации, относящейся к разным задачам, осуществляемым в пакете.

Диалоговое окно Set Values

С помощью диалогового окна Set Values можно задавать исходные значения любых свойств этапа прогона, используемых в пакете (в описанном выше простом пакете такие свойства не применялись).

Диалоговое окно Verification

Под одинаковым именем файла могут быть записаны совершенно разные пакеты (при условии, что они находятся в разных каталогах файловой системы). Кроме того, допускается возможность хранить в файловой системе или хранилище пакетов несколько версий одного и того же пакета. Диалоговое окно Verification предназначено для решения задачи поиска по заданным критериям или проверки имени и (или) версии пакета, который должен быть вызван на выполнение.

Диалоговое окно Command Line

Диалоговое окно Command Line позволяет воспользоваться возможностью вызывать пакеты SSIS на выполнение из командной строки (такая возможность является удобной, например, при осуществлении попытки вызвать пакеты DTS на выполнение из пакетного файла). С помощью диалогового окна Command Line программы SSIS Package Execution Utility можно задать параметры, которые должны использоваться при вызове пакета на выполнение из командной строки.

Программа Package Execution Utility определяет большинство параметров прогона пакета автоматически, а данное диалоговое окно позволяет уточнить значения опций, которые вступят в действие после того, как вы дадите распоряжение приступить к выполнению пакета, щелкнув на кнопке Execute.

Выполнение пакета

Для того чтобы выполнить запуск и прогон пакета, необходимо щелкнуть на кнопке Execute в окне Package Execution Utility. После завершения работы пакета текстовый файл должен находиться в том каталоге, в котором было предусмотрено его сохранение при определении пакета. Откройте этот файл, ознакомьтесь с ним и убедитесь в том, что он соответствует вашим требованиям.

Вызов пакета на выполнение с помощью программы Business Intelligence Development Studio

Возможность редактирования пакета предоставляется также с помощью программы Business Intelligence Development Studio. Эта программа позволяет либо вызвать весь пакет на выполнение (щелкнув для его прогона на кнопке Execute, как и при запуске любого другого приложения Visual Studio), либо открыть пакет для редактирования, дважды щелкнув на нем в окне Solution Explorer (после этого достаточно щелкнуть правой кнопкой мыши на любом конкретном элементе, который должен быть выполнен)

Вызов пакета на выполнение с помощью программы Management Studio

Безусловно, в программе Management Studio редактирование пакетов не предусмотрено, но эта программа предоставляет возможность вызывать пакеты на выполнение. В окне Registered Servers программы Management Studio щелкните на пиктограмме Integration Services, а затем дважды щелкните на названии сервера, на котором необходимо выполнить интересующий вас пакет (для этого может потребоваться зарегистрировать данный сервер в качестве сервера Integration Services в программе Management Studio). В результате должно быть создано соединение со службами Integration Services на этом сервере, а в окне Object Explorer должен быть добавлен узел Integration Services.

Чтобы иметь возможность вызвать пакет на выполнение таким образом (с использованием программы Management Studio), необходимо определить пакет как локальный по отношению к серверу (а не находящийся в файловой системе). К счастью, после щелчка правой кнопкой мыши на узле File System под элементом Stored Packages предоставляется возможность осуществить импорт пакета с помощью СУБД SQL Server. Достаточно перейти в файловой системе к созданному пакету, присвоить ему имя, предназначенное для применения в хранилище пакетов, а затем импортировать. После этого появляется возможность щелкать правой кнопкой мыши на обозначении пакета и вызывать его на выполнение в любое время. (При этом вызывается утилита запуска на выполнение, описанная в предыдущем разделе, поэтому исключаем дальнейшее описание данной темы.)

Редактирование пакета

На этом описание средств SSIS практически завершается, но следует отметить, что в настоящей главе даны лишь краткие наметки всего того, что следует освоить при изучении этого программного обеспечения.

В настоящем разделе кратко описана процедура редактирования пакетов. Точно такая же процедура применяется для создания новых пакетов, но, поскольку данная книга предназначена для начинающих, постараемся упростить изложение, взяв за основу пакет, который уже почти полностью готов к работе. Это описание не дает возможности освоить все средства SSIS на уровне специалиста, но позволяет получить представление о том, как формируются относительно сложные пакеты.

Откройте программу Business Intelligence Development Studio, выберите пункт меню File⇨ New Project и создайте новый проект Integration Services Project (рис. 18.10).

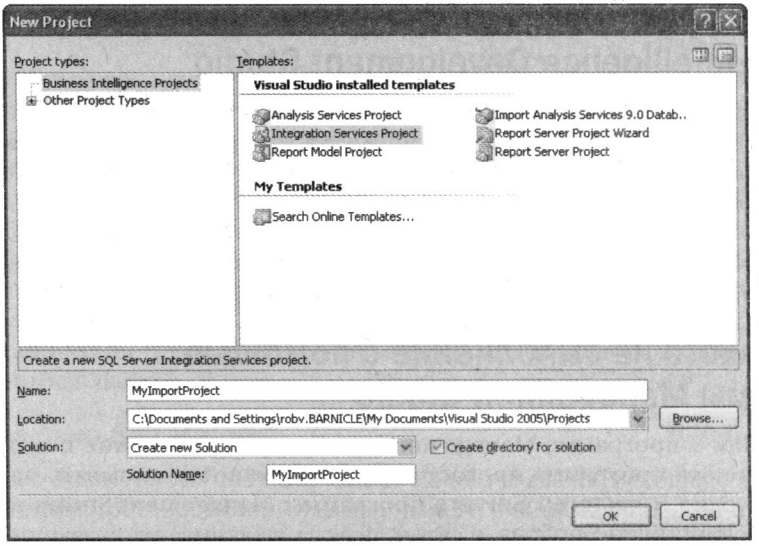


***Рис. 18.10. Создание нового проекта* Integration Services Project *в диалоговом окне* New Project**

В открывшемся окне New Project присвойте имя новому проекту. На рис. 18.10 показан пример, в котором проекту присвоено имя MyImportProject.

После щелчка на кнопке OK создается рабочее пространство приложения. Обратите внимание на то, что после этого автоматически создается пустой пакет с именем `Package.dtsx`. Но в данном случае задача состоит в том, чтобы отредактировать, а не создать пакет, поэтому щелкните правой кнопкой мыши на названии нового пакета и выберите команду Delete, чтобы удалить его. Затем щелкните правой кнопкой мыши на узле SSIS Packages и выберите команду Add Existing Package. Откроется диалоговое окно с предложением указать, откуда должен быть взят пакет; напомним, что для хранения пакетов SSIS предусмотрено много возможностей. В этом случае рассматривается пакет, для хранения которого выбрана файловая система, поэтому в диалоговом окне Add Copy of Existing Package в поле Package location установлено значение File System (рис. 18.11).

Таким образом, в окне, показанном на рис. 18.11, в качестве источника пакета выбрана файловая система, File System, а затем было заполнено поле Package path, которое находится в нижней части окна и указывает местонахождение файла в файловой системе. Путь, указанный в этом поле, содержит информацию о том, где находится файл пакета, относящийся к проекту экспорта.

После этого название пакета должно быть добавлено к узлу SSIS Packages в окне Solution Explorer; щелкните правой кнопки мыши на названии пакета и выберите команду Open. В результате на редактирование будет вызван относительно простой пакет, содержащий только одну задачу (рис. 18.12).

Имя задачи Data Flow Task представляет собой еще одно обозначение работы, выполняемой пакетом. В данном случае в состав проекта экспорта фактически входит только одна задача, но можно дважды щелкнуть на этом имени правой кнопкой мыши, чтобы раскрыть содержание этой задачи и рассмотреть, какие компоненты входят в данный пакет SSIS (рис. 18.13).

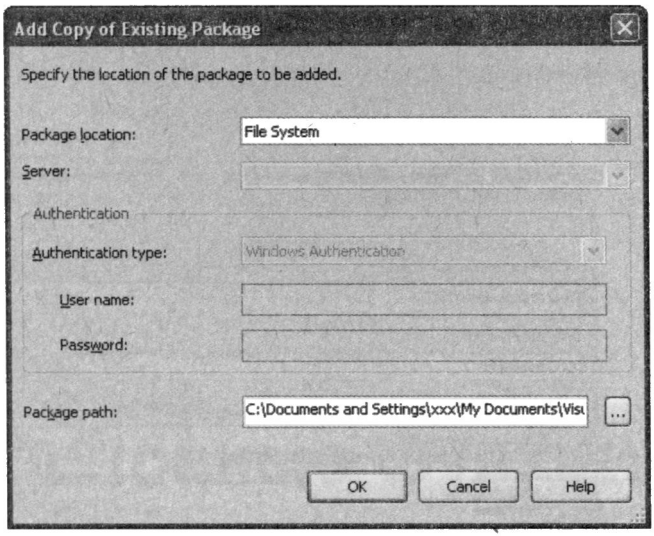


Рис. 18.11. Диалоговое окно Add Copy of Existing Package

Рис. 18.12. Задача Data Flow Task

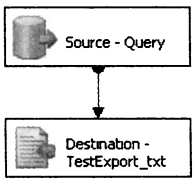


Рис. 18.13. Компоненты пакета SSIS

Рассматриваемая задача экспорта данных состоит из двух шагов. Вначале выполняется запрос для получения исходных данных, затем данные записываются в текстовый файл. Обратите внимание на то, что на рис. 18.13 показана стрелка, которая имеет особую форму и обозначает направление потока данных. Дважды щелкните кнопкой мыши на узле Source - Query, чтобы открыть исходный запрос (рис. 18.14).

Итак, очевидно, что при использовании программы-мастер Import/Export Wizard разработчик может не заниматься отдельно созданием каждого компонента задачи экспорта или импорта данных; это позволяет (в большей или меньшей степени) устранить значительную часть сложностей, связанных с созданием пакетов. С другой стороны, программа Business Intelligence Development Studio дает возможность непосредственно создавать собственные пакеты, определяя требуемые параметры настройки и объекты.

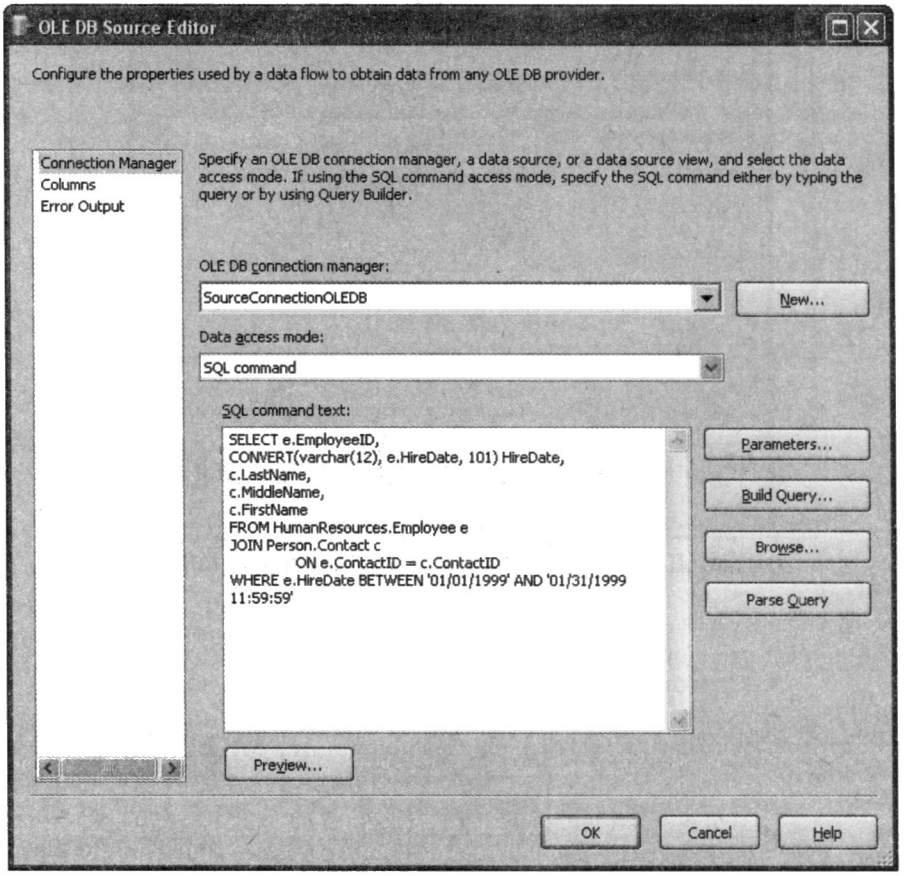


Рис. 18.14. Окно OLE DB Source Editor

Таким образом, программа Business Intelligence Development Studio позволяет создавать пакеты с нуля, а также редактировать пакеты, созданные с помощью программы-мастера, как в приведенном выше примере.

Резюме

Службы Integration Services, которые входят в состав программного обеспечения SQL Server, представляют собой надежное инструментальное средство извлечения, преобразования и загрузки данных. В частности, службы Integration Services могут использоваться для осуществления единоразового или многократного импорта и экспорта данных из базы данных и обратно, используя при этом различные сочетания источников данных.

Глубокое усвоение возможностей Integration Services приносит огромную пользу, поскольку позволяет получить столь важные навыки импорта и экспорта. Приступая к освоению этих навыков, начинайте с простых задач, а затем постепенно усложняйте выполняемые операции. По мере изучения службы SSIS обращайтесь к другим книгам, которые содержат более подробное описание SSIS.

19

Основные функции администратора

Предыдущие главы в основном касались проблематики разработки баз данных, а в настоящей главе речь пойдет о сопровождении и администрировании разработанных баз данных.

Некоторые разработчики считают, что задачи обслуживания базы данных относятся исключительно к сфере функций администратора баз данных. В связи с этим иногда возникает такая ситуация, что разработчики главным образом сосредоточиваются на создании приложений для баз данных и отказываются заниматься какими-либо вопросами, которые относятся к области эксплуатации этих приложений.

Но следует отметить, что любые приложения, действующие под управлением базы данных, коренным образом отличаются от большинства автономных приложений. Дело в том, что автономные приложения обычно либо не требуют сопровождения, либо работают только с отдельными файлами, которые можно относительно легко подготовить для работы или по истечении определенного времени переписать в какой-то другой каталог для создания резервной копии. Кроме того, при эксплуатации автономных приложений обычно не приходится решать такие сложные задачи сопровождения, как при использовании базы данных.

В настоящей главе приведены некоторые рекомендации, позволяющие не только успешно решать основные задачи сопровождения в целях обеспечения бесперебойной работы базы данных, но и получить возможность успешно устранять возникающие нарушения в работе и аварийные ситуации.

Основные темы, рассматриваемые в данной главе, перечислены ниже.

- Планирование заданий.
- Резервное копирование и восстановление.

- Выполнение основных задач по устранению фрагментации и перестройке индексов.
- Использование предупреждающих сообщений.
- Архивирование.

Безусловно, вышеназванные функции далеко не исчерпывают круг задач администрирования, но они представляют собой своего рода перечень минимальных требований, которые, как правило, приходится учитывать, планируя внедрение приложения в эксплуатацию.

Поскольку настоящая книга предназначена для начинающих, в этой главе для решения задач сопровождения в основном применяются графические инструментальные средства, которые входят в состав программного обеспечения SQL Server Management Studio. Но необходимо отметить, что пользователи, имеющие хорошую подготовку, могут воспользоваться также такими программными средствами, которые позволяют осуществлять многие функции администрирования программным путем, с помощью объектной модели SQL Server Management Objects (SMO).

Планирование заданий

Многие задачи сопровождения, которые рассматриваются в настоящей главе, могут быть **запланированы**. Планирование заданий позволяет обеспечить выполнение задач, создающих большую нагрузку на систему, в нерабочие часы. Кроме того, планирование заданий гарантирует, что по забывчивости не будет пропущено время выполнения ответственной операции. Ведь время от времени приходится слышать настораживающие сообщения, что в какой-то компании произошел серьезный сбой, связанный с тем, что не была вовремя выполнена перестройка индексов или не создана резервная копия из-за того, что кто-то "забыл" это сделать или посчитал, что соответствующее задание запланировано, но так и не проверил, так ли это на самом деле.

Если вы получили хорошую подготовку в области эксплуатации программного обеспечения Windows Server и привыкли планировать другие задания с помощью службы Windows Scheduler, то можете использовать и эти средства планирования для поддержки СУБД SQL Server. Безусловно, планирование всех заданий с помощью службы Windows Scheduler позволяет связать всю работу по планированию с одним приложением, но следует учитывать, что СУБД SQL Server поддерживает некоторые более совершенные средства выполнения по условию.

При планировании работы обычно приходится сталкиваться с двумя основными понятиями, определяющими единицы работы, — с заданиями и задачами. Определения этих понятий приведены ниже.

- **Задача.** Это отдельный процесс, который должен быть выполнен, или пакет команд, прогон которого должен быть осуществлен. Задачи не являются независимыми и могут рассматриваться только как компоненты заданий.
- **Задание.** Представляет собой группу из одной или нескольких задач. которые должны выполняться в связи друг с другом. Кроме того, предусмотрена возможность устанавливать зависимости и определять дальнейший ход выполнения с учетом успешного или неудачного завершения отдельных задач (если, напри-

мер, задача A должна выполняться при условии успешного завершения предыдущей задачи, а в случае неудачного завершения предыдущей задачи должна быть выполнена задача B.)

Планирование заданий может осуществляться с учетом описанных ниже условий.

- Ежедневно, еженедельно или ежемесячно.
- В определенное время суток.
- С заданной частотой (допустим, через каждые 10 минут или через каждый час).
- После того как процессор простаивает в течение указанного времени.
- После запуска программы SQL Server Agent.
- В ответ на предупреждающее сообщение.

Запуск задач на выполнение осуществляется только в силу того, что они являются частью задания, а последовательность запуска определяется с учетом правил перехода от одной задачи к другой, которые установлены для задания. Из того, что произошел запуск задания, не следует, что будет выполнен прогон всех задач, входящих в состав этого задания; одни задачи могут быть выполнены, а другие — нет, поскольку ход выполнения зависит от успешного или неудачного завершения предыдущих задач в задании и от установленных **правил перехода**. Программное обеспечение SQL Server не только позволяет автоматически активизировать одну задачу после завершения другой, но и обеспечивает возможность выполнения какой-то совсем другой работы (допустим, осуществление некоторой разновидности задачи восстановления), если текущая задача окончится неудачей.

Предусмотрена возможность не только дать программному обеспечению SQL Server указания по осуществлению перехода от одной задачи к другой, но и указать, что должны быть выполнены описанные ниже действия.

- Передача уведомлений об успешном или неудачном завершении задания оператору. Обеспечивается возможность передавать каждое отдельное извещение в виде сетевого сообщения (которое всплывает на экране пользователя, при условии, что он работает в системе), отправлять сообщение на пейджер, а также посылать письмо по адресу электронной почты каждого оператора.
- Запись информации о результатах выполнения задания в файл регистрации событий.
- Автоматическое удаление задания (для предотвращения его выполнения в дальнейшем), а также, вообще говоря, исключение всех объектов, ставших ненужными.

В следующем разделе кратко описано, как создать учетную запись оператора в программе Management Studio.

Создание учетной записи оператора

Чтобы иметь возможность использовать средства передачи извещений программы SQL Agent, необходимо создать учетную запись оператора и ввести все необходимые сведения о том, кому должны передаваться извещения. Часть настройки конфигурации средств передачи извещений, связанная с созданием учетной записи оператора, как правило, не может быть осуществлена с помощью какого-либо автоматизированного процесса или предусмотрена в составе разрабатываемого кода, поэто-

му такие учетные записи обычно создаются администратором базы данных вручную. Ниже приведено краткое описание процедуры создания учетной записи оператора, поскольку это описание позволяет проще понять процедуру планирования заданий.

Создание учетной записи оператора в программе Management Studio

Для создания учетной записи оператора с помощью программы Management Studio необходимо перейти в окне SQL Server Agent к узлу, обозначающему сервер, для которого должна быть создана учетная запись оператора. Разверните этот узел SQL Server Agent, щелкните правой кнопкой мыши на элементе Operators и выберите команду New Operator.

Следует учитывать, что в зависимости от конкретной инсталляции запуск службы SQL Server Agent может не осуществляться автоматически по умолчанию. Если вам придется столкнуться с какими-либо проблемами или вы обнаружите, что на пиктограмме SQL Server Agent в окне Management Studio имеется небольшой красный квадрат, то, по-видимому, указанная служба настроена на запуск вручную или даже отменена; в таком случае, скорее всего, потребуется внести такие изменения, чтобы запуск этой службы осуществлялся автоматически. В любом случае убедитесь в том, что служба SQL Server Agent работает, поскольку иначе вы не сможете проверить примеры, выполнить практические задания и упражнения, приведенные в этой главе. Для этого необходимо щелкнуть правой кнопкой мыши на узле Agent и выбрать команду Manage Service.

Откроется диалоговое окно New Operator, показанное на рис. 19.1 (часть полей на этом рисунке уже заполнена).

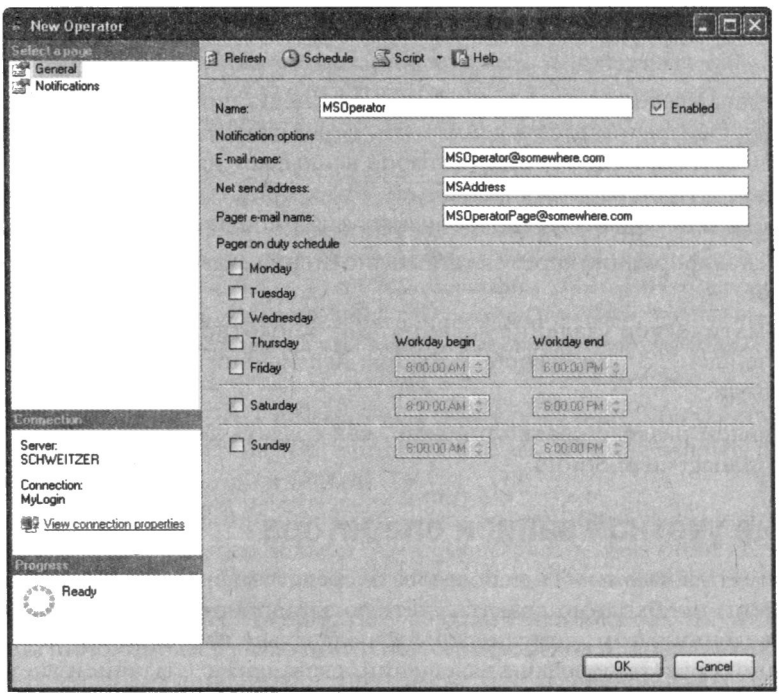


Рис. 19.1. Диалоговое окно New Operator

Теперь мы можем приступить к заполнению расписания, позволяющего указать, в какое время оператор должен получать по электронной почте уведомления о возникновении ошибок определенных типов; для этого используется вкладка Notifications.

Поскольку речь идет о вкладке Notifications, щелкните на обозначении Notifications и откройте эту вкладку (рис. 19.2).

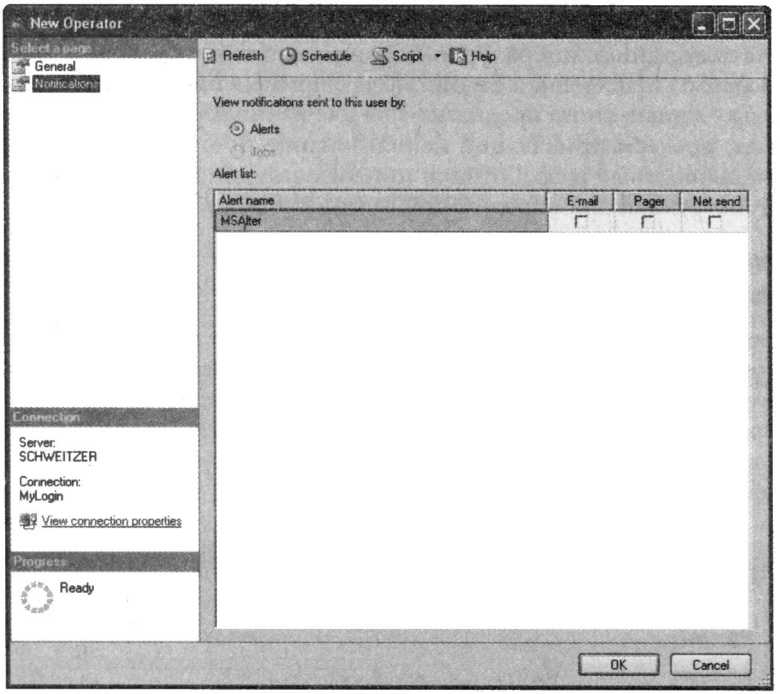


Рис. 19.2. Вкладка Notifications

До тех пор пока в системе не определен достаточно большой набор предупреждающих сообщений (дополнительная информация на эту тему приведена ниже в данной главе), страница, показанная на рис. 19.2, не представляет особого интереса. Эта страница предназначена исключительно для определения того, какие извещения должен получать оператор в зависимости от активизации определенных предупреждающих сообщений. Еще раз отметим, что все нюансы настройки системы передачи извещений трудно понять, не изучив, как формируются предупреждающие сообщения, но достаточно отметить, что предупреждающие сообщения активизируются при возникновении некоторых событий в базе данных, а страница Notifications позволяет указать, какие предупреждающие сообщения получает конкретный оператор.

Определение заданий и задач

Как уже было сказано, задания представляют собой коллекции, состоящие из одной или нескольких задач. А задача – это логическая единица работы, обеспечивающая, например, резервное копирование одной базы данных или прогон одного сценария T-SQL, предназначенного для реализации какой-то конкретной потребности, такой как перестройка всех индексов.

Даже несмотря на то, что задание может состоять из нескольких задач, нет никаких гарантий, что будет выполнена каждая задача в каком-то конкретном задании. Запуск отдельных задач или отказ от запуска происходит с учетом того, насколько удачно выполнены другие задачи в этом задании, а также в зависимости от того, какие действия определены в качестве реакции на каждый случай успеха или неудачи. Например, можно даже отменить выполнение остальной части задания при условии, что одна из задач окончится неудачей.

Как и учетные записи операторов, задания могут быть созданы не только с помощью программы Management Studio, но и с применением программных конструкций. Поскольку данная книга предназначена для начинающих, в ней рассматривается только метод, предусматривающий использование программы Management Studio (но даже очень опытные разработчики приложений для SQL Server применяют программные конструкции для определений заданий и задач исключительно редко).

Определение заданий и задач с использованием программы Management Studio

Программа SQL Server Management Studio позволяет чрезвычайно просто создавать планируемые задания. Для этого достаточно перейти к узлу SQL Server Agent, относящемуся к используемому серверу, щелкнуть правой кнопкой мыши на элементе Jobs и выбрать команду New Job. В результате откроется диалоговое окно New Job (рис. 19.3), в котором имеется несколько вкладок, что дает возможность обеспечить поэтапное формирование задания.

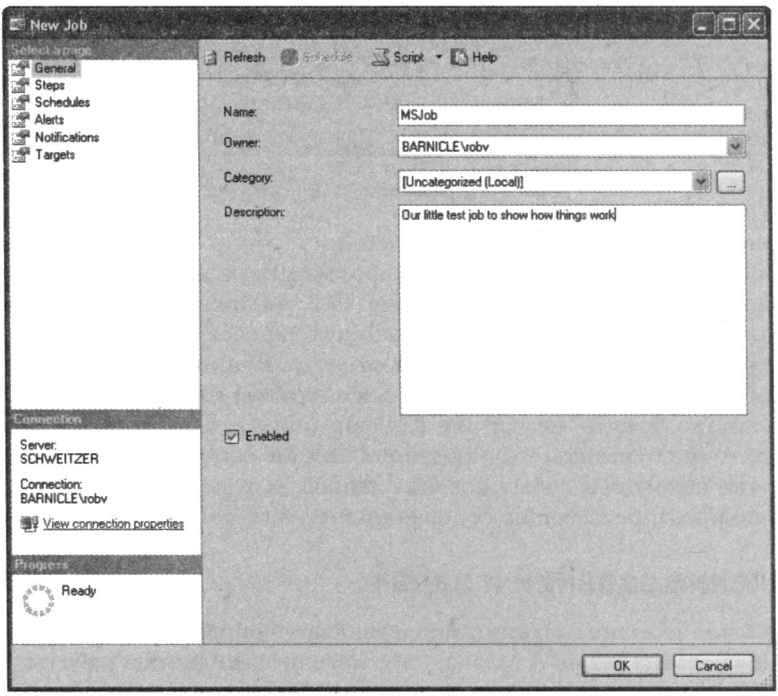


Рис. 19.3. Диалоговое окно New Job

В поле Name (см. рис. 19.3) можно ввести произвольное имя задания. Единственное требование к формату этого имени состоит в том, что оно должно соответствовать правилам именования объектов SQL Server, которые были приведены выше в данной книге.

Почти все прочие значения полей (см. рис. 19.3), кроме Category (Категория), опять-таки не требуют пояснений. Категория — это всего лишь один из способов группирования заданий. Как правило, многие задания, характерные для конкретного приложения, соответствуют неопределенной категории, которая обозначается как Uncategorized, но возможны также ситуации, когда потребуется определить задание, относящееся к более конкретной категории, — Web Assistant, Database Maintenance, Full Text или Replication. В подобных случаях задания распределяются по отдельным категориям, для того чтобы было проще определить их назначение.

Закончив работу с вкладкой, показанной на рис. 19.3, перейдем к вкладке Steps (рис. 19.4). Именно с этой вкладки начинается ввод информации, позволяющей программному обеспечению SQL Server приступить к созданию новых задач, которые должны войти в состав задания.

Для того чтобы ввести в задание новый шаг (новую задачу), щелкните на кнопке New и заполните поля в диалоговом окне New Job Step (рис. 19.5). В данном примере применим оператор T-SQL для активизации фиктивной ошибки исключительно ради того, чтобы можно было ознакомиться с тем, какие действия будут осуществляться после планирования этого задания. Но следует отметить, что слева от окна Command имеется кнопка Open, с помощью которой можно импортировать в это окно сценарии SQL, хранящиеся в файлах.

Продолжим работу и перейдем к вкладке Advanced (рис. 19.6) диалогового окна Job Step Properties (см. рис. 19.5). Именно в этом окне можно ознакомиться с некоторыми из наиболее привлекательных функциональных средств, поддерживаемых планировщиком заданий.

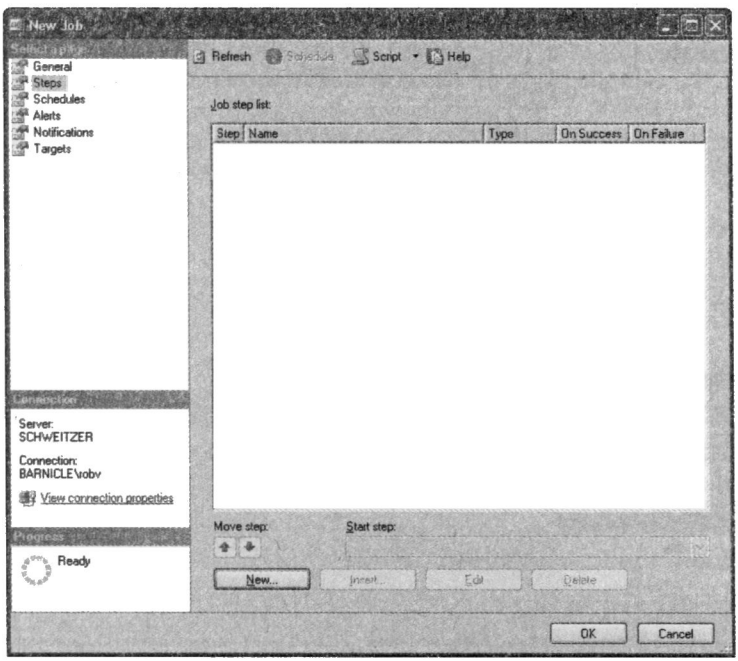


Рис. 19.4. Вкладка Steps

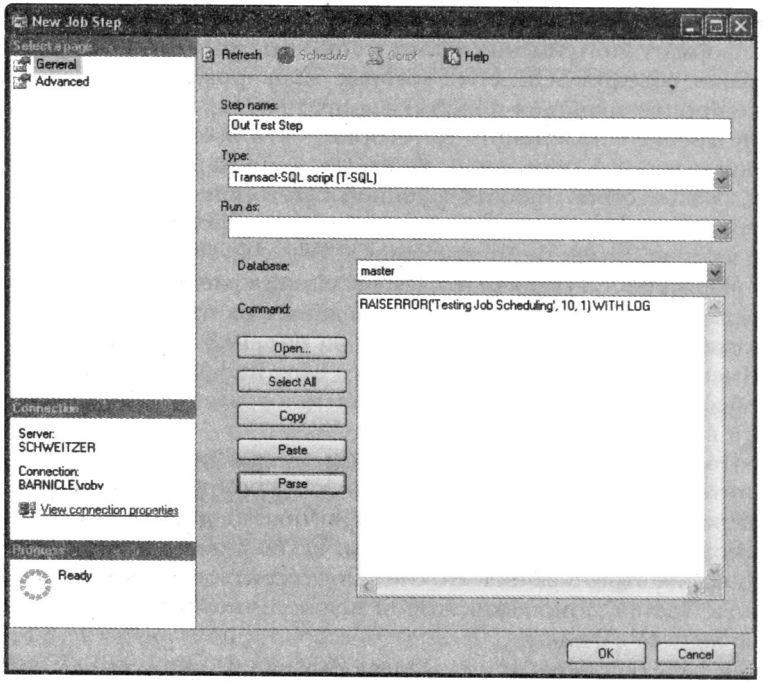


Рис. 19.5. Диалоговое окно New Job Step

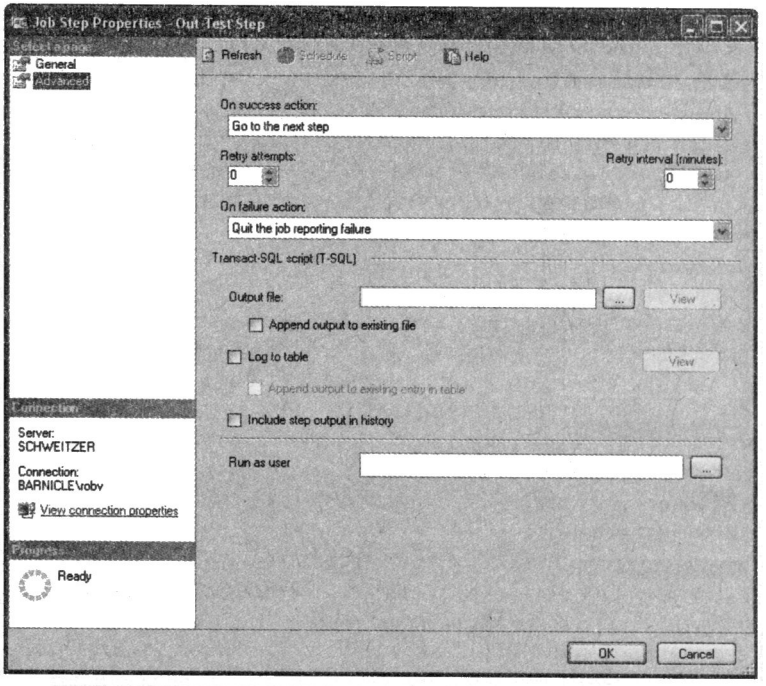


Рис. 19.6. Вкладка Advanced

В этом диалоговом окне заслуживают внимания несколько описанных ниже особенностей.

- Это окно позволяет автоматически настраивать задание на повторный запуск через указанный интервал времени в случае неудачного завершения одной из задач.
- Может быть указано, что делать в случае успешного или неудачного завершения задания; в частности, применительно к каждому результату (успешному или неудачному) можно предусмотреть следующее:
  - отказаться от активизации сообщения об успешном завершении задания;
  - отказаться от активизации сообщения о неудачном завершении задания;
  - перейти к следующему шагу.
- Можно вывести результаты в файл (такая возможность очень удобна для проведения аудита).
- Может быть предусмотрено обеспечение анонимности любого пользователя (что позволяет проще решать задачи предоставления прав доступа). Следует отметить, что данной возможностью разрешается пользоваться только тем, кто имеет право определять привилегии для такого пользователя. Дело в том, что вход в систему в качестве системного администратора дает возможность выполнять задания от имени владельца базы данных (`dbo`) или почти любого другого пользователя. С другой стороны, в распоряжении обычного пользователя, по-видимому, имеется только гостевая учетная запись (если он не является владельцем базы данных), но ведь в большинстве случаев обычному пользователю не предоставляется и возможность планировать на выполнение собственные задания (как правило, такая возможность предоставляется исключительно в рамках клиентского приложения).

Итак, маловероятно, что неудачей завершится сама попытка выполнения оператора `RAISERROR` (см. рис. 19.5), поэтому достаточно выбрать в окне (см. рис. 19.6) заданное по умолчанию значение Quit the job reporting failure (Отказаться от передачи сообщения о неудачном завершении задания). Другие варианты действий, осуществляемых в случае неудачного завершения задания, будут описаны ниже, когда речь пойдет о резервном копировании.

После этого снова перейдем к основному диалоговому окну New Job. На данном этапе можно перейти к узлу Schedules (рис. 19.7).

Диалоговое окно, приведенное на рис. 19.7, позволяет определить одно или несколько запланированных значений времени для запуска этого задания на выполнение. Чтобы создать задание, щелкните на кнопке New. В результате откроется еще одно диалоговое окно, New Job Schedule (рис. 19.8).

Диалоговое окно, приведенное на рис. 19.8, уже почти полностью заполнено (чтобы больше не приходилось загромождать эту главу копиями экрана), но достаточно отметить, что в этом диалоговом окне приведено все необходимое для создания нового расписания запуска рассматриваемого задания. В частности, в этом окне показано, как часто должно выполняться это задание и сколько раз оно должно повторяться.

При определении частоты и кратности выполнения задания может возникнуть путаница, поскольку для этого в диалоговом окне New Job Schedule применяются немного непривычные опции. В частности, если требуется, чтобы задание выполнялось несколько раз ежесуточно, то в поле Occurs нужно задать значение Daily, а в поле Recurs выбрать значение 1 day(s).

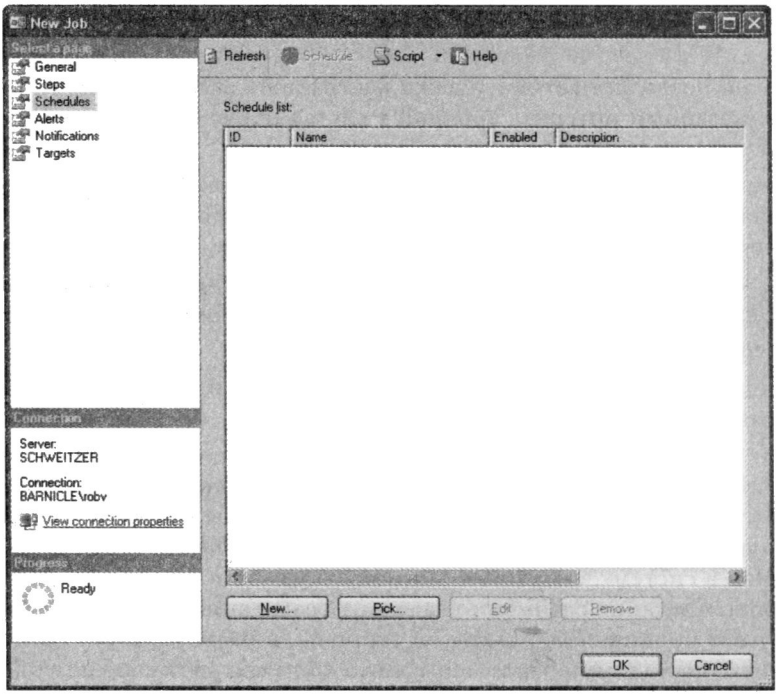


Рис. 19.7. Вкладка Schedules

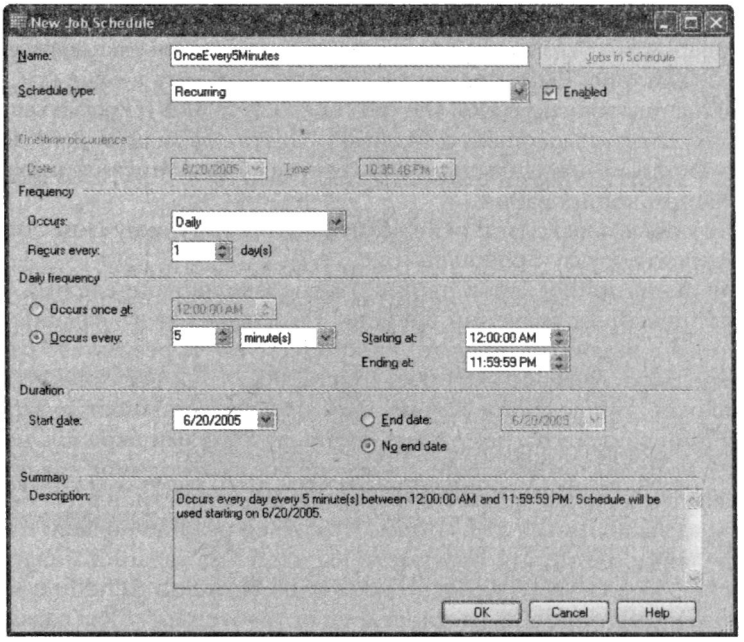


Рис. 19.8. Диалоговое окно New Job Schedule

Создается впечатление, что в результате этого задание планируется на выполнение только один раз в сутки, но остается еще возможность указать с помощью переключателя с элементами Occurs once at и Occurs every, должно ли задание быть выполнено единожды или оно должно выполняться повторно через определенный интервал. В рассматриваемом случае предусмотрен запуск задания на выполнение через каждые 5 минут.

Теперь можно перейти к следующему узлу с описанием свойств задания. Речь идет о вкладке Alerts, которая показана на рис. 19.9.

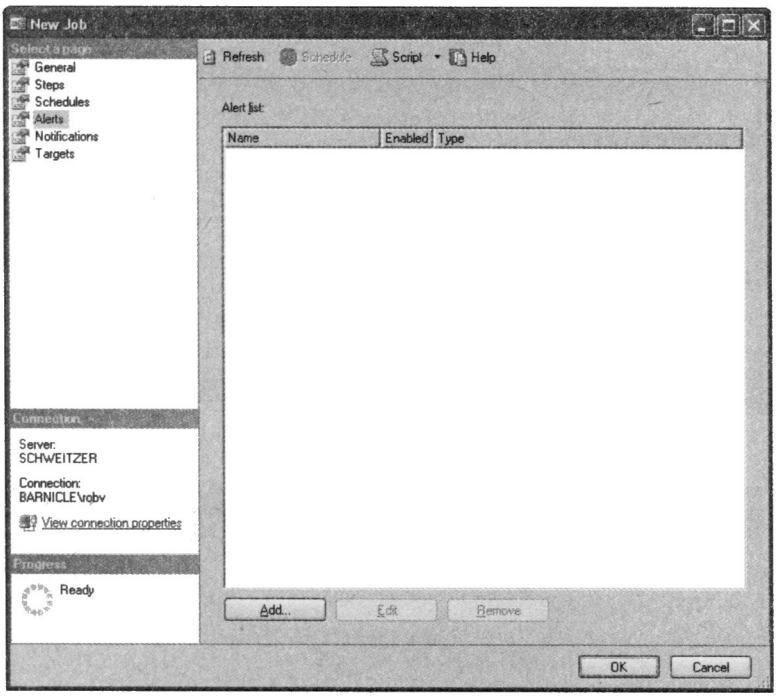


Рис. 19.9. Вкладка Alerts

Диалоговое окно, приведенное на рис. 19.9, позволяет выбрать предупреждающие сообщения, которые могут быть сформированы с учетом происходящих событий. Щелкните на кнопке Add, чтобы открыть еще одно содержательное диалоговое окно, New Alert (рис. 19.10).

Первый узел, General, в левой части окна (см. рис. 19.10) позволяет определить основные свойства предупреждающего сообщения. Например, с его помощью можно ограничиться формированием извещений, относящихся только к одной конкретной базе данных. Кроме того, в поле Severity можно указать, насколько серьезными должны стать сложившиеся условия, прежде чем активизируется предупреждающее сообщение (под этим подразумевается степень серьезности ошибки).

Закончив работу с этим диалоговым окном, перейдем к странице, показанной на рис. 19.11, которая открывается после щелчка на узле Response.

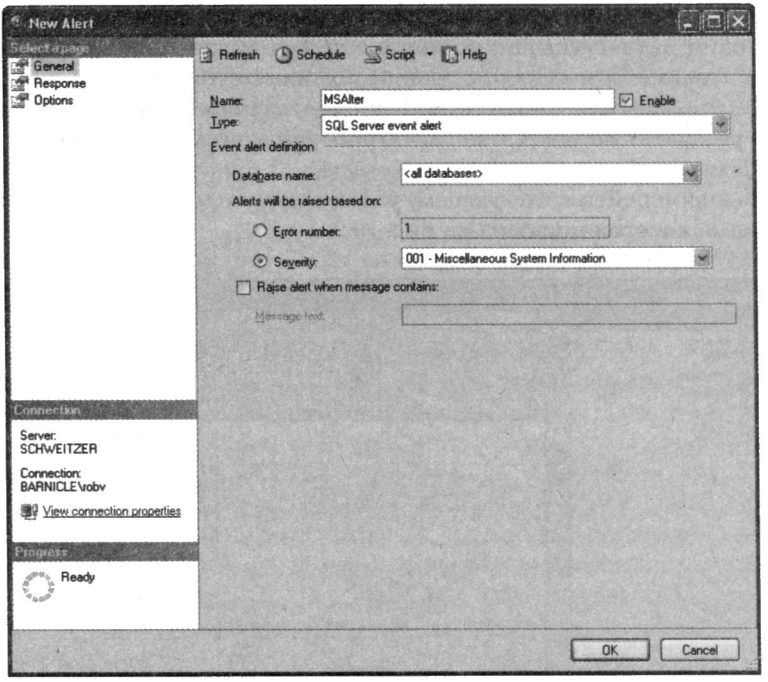


Рис. 19.10. Диалоговое окно New Alert

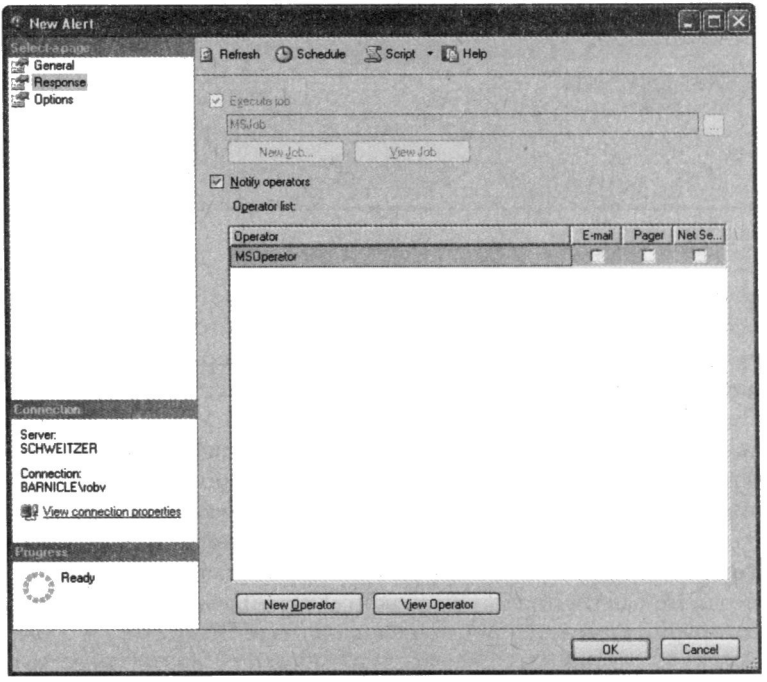


Рис. 19.11. Страница Response

Обратите внимание на то, что окно, приведенное на рис. 19.11, позволяет указать в качестве адресата предупреждающих сообщений оператора, учетная запись которого была создана выше в данной главе. Программное обеспечение SQL Server Agent определяет, по какому адресу электронной почты или адресу сетевой рассылки необходимо передавать извещения, именно на основании определения учетной записи оператора. Следует также отметить, что справа от имени учетной записи оператора имеются флажки, которые позволяют взять на себя контроль над тем, каким путем должны передаваться извещения операторам.

Наконец отметим, что после щелчка на узле Options (см. рис. 19.11) откроется страница Options (рис. 19.12).

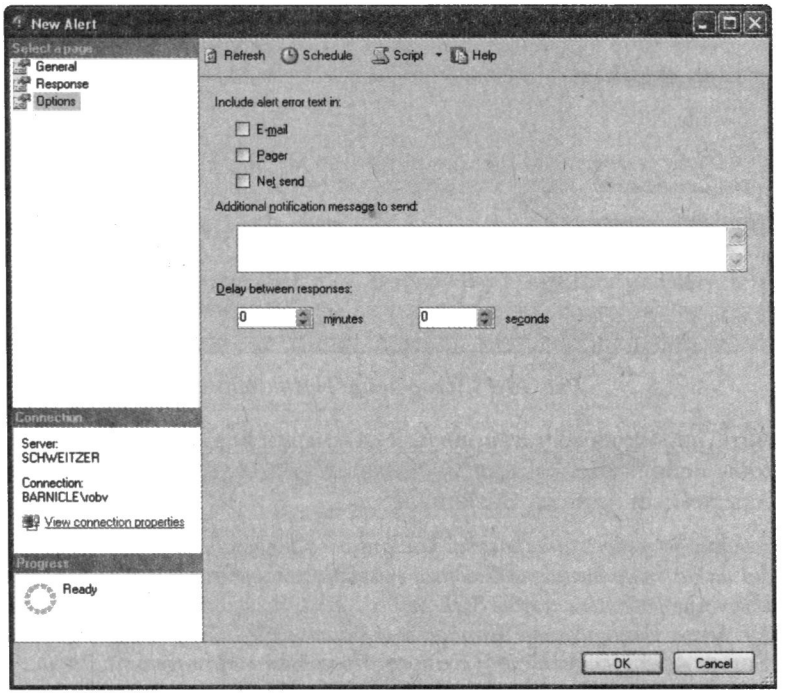


Рис. 19.12. Страница Options

Завершив выполнение описанных операций, вернемся в главное диалоговое окно New Job и откроем щелчком на узле Notifications одноименную страницу (рис. 19.13).

Страница Notifications (см. рис. 19.13) позволяет отказаться от применявшейся ранее модели формирования предупреждающих сообщений и определить формат отклика, более подходящий для конкретного задания; в настоящей главе мы ограничимся только приведенным выше описанием и лишь отметим, что это диалоговое окно позволяет определить форматы конкретных дополнительных извещений.

Теперь можно щелкнуть на кнопке OK и выйти из диалогового окна New Job. После этого придется подождать несколько минут, прежде чем активизируется задание (записи журнала появляются через каждые пять минут в **журнале событий Windows**). Для того чтобы просмотреть новые записи журнала, выберите элемент меню Start⇨Programs⇨Administrative Tools⇨Event Viewer.

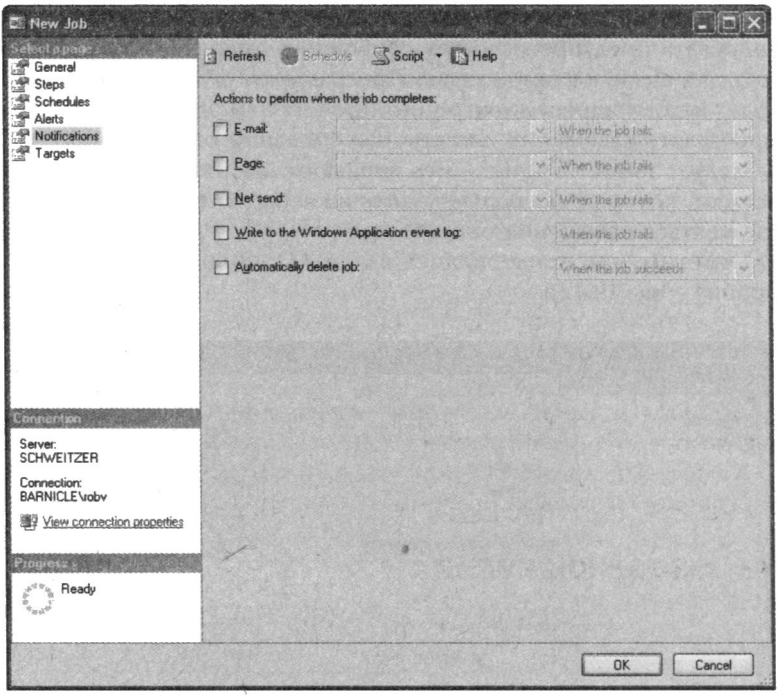


Рис. 19.13. Страница Notifications

Кроме того, необходимо установить в открывшемся окне переключатель таким образом, чтобы использовался журнал приложений, Application log, а не заданный по умолчанию системный журнал, System log.

Если вы собираетесь эксплуатировать запланированные задания, подобные описанному, то для запуска их на выполнение необходимо вызвать программу SQL Server Agent. Чтобы проверить состояние работы службы SQL Server Agent, можно запустить на выполнение программу SQL Server Configuration Manager и выбрать службу SQL Server Agent или перейти к узлу SQL Server Agent в окне Object Explorer программы Management Studio.

Кроме того, после проверки рассматриваемого задания не забудьте его отменить. (Щелкните правой кнопкой мыши на имени этого задания в программе Management Studio после того, как убедитесь, что оно работает в соответствии с ожиданиями.) В противном случае задание будет вызываться повторно снова и снова и создавать записи в журнале приложений; в конечном итоге журнал приложений заполнится и в ходе эксплуатации системы возникнут проблемы.

Резервное копирование и восстановление

Недопустимо, чтобы какое-либо приложение, действующее под управлением базы данных, передавалось в эксплуатацию или поставлялось заказчику без определенного механизма резервного копирования и восстановления. Можно на полном основании утверждать, что процентная доля операций с базами данных, в которых не применяются хоть какие-то приемлемые средства резервного копирования, является буквально поразительной. Иными словами, положение очень серьезное.

Обдумывая вопрос об организации резервного копирования, необходимо прежде всего руководствоваться таким простым правилом: оно должно осуществляться заблаговременно и часто. Из этого следует, что недостаточно лишь создать копию некоторого файла на том же диске и забыть о ее существовании; прежде всего необходимо обеспечить, чтобы требуемая копия была перенесена полностью на отдельный носитель (в идеальном случае даже на другую площадку), чтобы гарантировать ее безопасность. В процессе эксплуатации случается всякое. Автор сам был свидетелем того, как пожар охватил серверы (запах был ужасным, но не менее ужасным был испуг персонала). Никому не пожелаешь, чтобы в подобном случае резервные копии погибли в огне вместе с исходными данными.

Разработчикам приложений, еще на обладающим достаточно весомым опытом, по-видимому, следует полагаться на то, что инструментальными средствами резервного копирования и восстановления СУБД SQL Server будут пользоваться сами представители заказчика или администраторы узла заказчика, но даже в этом случае необходимо иметь достаточную подготовку для того, чтобы оказать заказчику помощь в случае необходимости. Кроме того, следует учитывать, что для разработчика непонимание потребностей заказчика является непростительным упущением.

Создание резервной копии

Фактически задача создания файла резервной копии для какой-то конкретной базы данных является относительно несложной. Для этого достаточно лишь перейти в окне Object Explorer к названию интересующей вас базы данных и щелкнуть на нем правой кнопкой мыши.

После этого выберите команды Tasks и Back up (рис. 19.14).

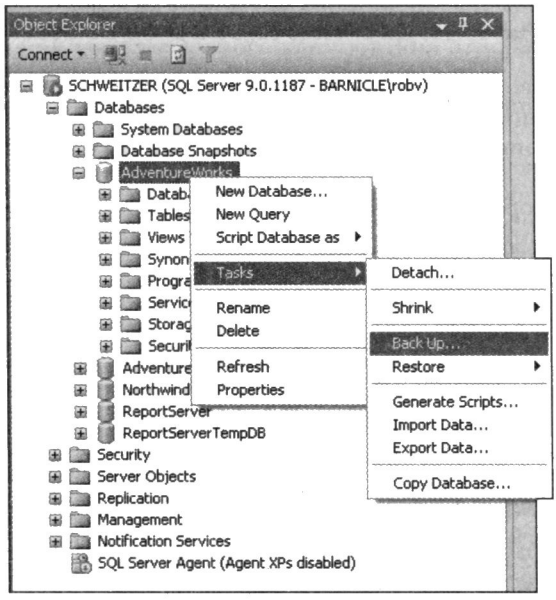


Рис. 19.14. Диалоговое окно Object Explorer, в котором выбраны команды Tasks и Back up

Откроется диалоговое окно, которое позволяет определить почти все необходимые параметры процесса резервного копирования (рис. 19.15).

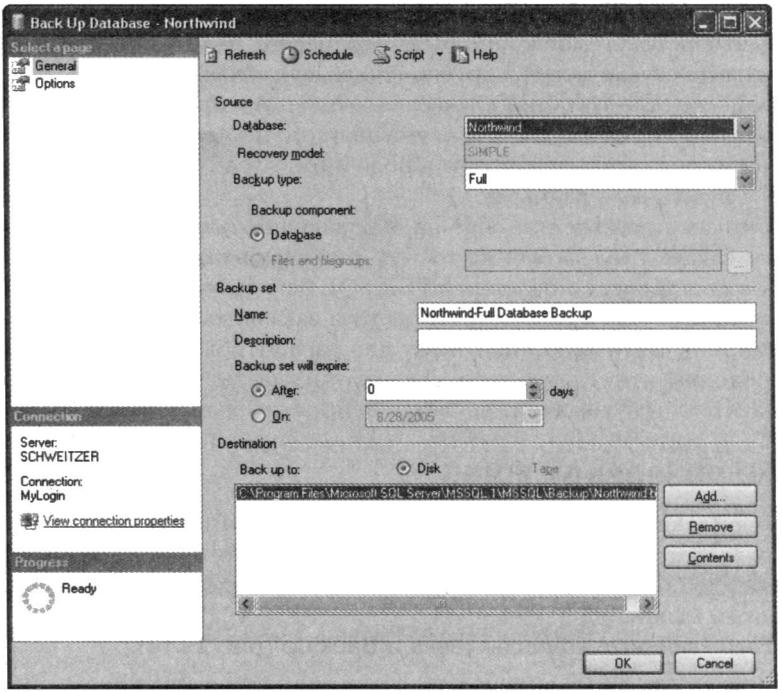


Рис. 19.15. Диалоговое окно Back Up Database

В диалоговом окне, показанном на рис. 19.15, первый параметр, Database, не требует пояснений, поскольку он указывает базу данных, для которой необходимо создать резервную копию. Но в дальнейшем приходится применять немного более сложные параметры.

Пока оставим в стороне описание других параметров, которые могут оказаться не совсем понятными, и рассмотрим параметр, обозначающий модель восстановления. Поле Recovery Model (см. рис. 19.15) содержит информацию о том, какой способ восстановления применяется для выбранной базы данных; этот параметр задается на уровне базы данных. Более подробное описание этого параметра приведено ниже; речь о нем также пойдет в следующем разделе в связи с описанием процесса резервного копирования журналов транзакций.

Описание остальных параметров является не столь сложным, но они должны рассматриваться с разбивкой по отдельным категориям.

Тип резервного копирования

Прежде всего необходимо выбрать тип резервного копирования, указав его в поле Backup type. Допускается использование двух или трех описанных ниже типов резервного копирования, количество которых зависит от заданной модели восстановления для базы данных (еще раз отметим, что дополнительные сведения о моделях восстановления приведены ниже).

- Full. Этот тип резервного копирования полностью соответствует своему названию. Он позволяет получить полную резервную копию существующих файлов базы данных в том состоянии, в котором они находились после фиксации последней транзакции, перед выполнением команды Backup.
- Differential. Резервное копирование этого типа может рассматриваться как создание резервной копии данных, изменившихся со времени получения предыдущей резервной копии. При получении дифференциальной резервной копии копируются только те экстенты (определение понятия экстент см. в главе 9), которые изменились со времени получения последней полной резервной копии. Как правило, процесс создания дифференциальной резервной копии осуществляется гораздо быстрее по сравнению с полным резервным копированием, а сама копия занимает меньше места. Объем дифференциальной резервной копии зависит от того, насколько многочисленными оказались изменения в самих данных. При эксплуатации очень крупных баз данных процесс создания резервных копий может занимать очень много времени, поэтому обычно принято использовать такую организацию работы, при которой полная резервная копия создается только один раз в неделю или даже один раз в месяц, а затем в промежутках между операциями полного резервного копирования создаются только дифференциальные резервные копии в целях экономии места и времени.
- Transaction Log. Этот параметр также полностью соответствует своему названию; он предусматривает получение копии журнала транзакций. Возможность использования этой опции предоставляется, только если в базе данных используются варианты ведения журнала Full (С включением всех операций) или Bulk (С включением неполных данных о массовых операциях). Если же используется вариант ведения журнала Simple, то опция Transaction Log становится недоступной. Более подробное описание этой темы также будет приведено немного позже.

К задаче определения типа резервного копирования относится также выбор компонента с помощью группы переключателей Backup component, подлежащего резервному копированию, но эта опция относится только к полным и дифференциальным резервным копиям.

В соответствии с назначением настоящей книги в этой главе речь должна идти в основном о резервном копировании всей базы данных (для чего используется переключатель Database). Тем не менее отметим также, что предусмотрена еще одна опция – Files and filegroups (Файлы и группы файлов). Напомним, что в главах этой книги вместе с описанием объектов базы данных и операторов создания баз данных кратко рассматривалось назначение файловых групп и отдельных файлов, предназначенных для хранения данных. Опция Files and filegroups позволяет выбрать только один файл (или группу файлов) резервной копии, но автор настоятельно рекомендует избегать этой опции тем пользователям SQL Server, которые еще не достигли уровня эксперта.

Следует еще раз подчеркнуть, что вы должны избегать использования данной опции до тех пор, пока не достигнете такого уровня, что вас можно будет считать почти что доктором наук в области резервного копирования для SQL Server. Эта опция имеет специальное назначение и должна обеспечивать ускоренное получение резервных копий для очень крупных инсталляций баз данных (измеряемых терабайтами), которые эксплуатируются в условиях высокой степени готовности. При создании и восстановлении резервных копий, полученных с помощью опции Files and filegroups, необходимо учитывать наличие серьезных проблем совместимости и брать на себя очень большую ответственность.

Набор резервного копирования

Область Backup set окна Back Up Database (см. рис. 19.15) содержит информацию об используемом наборе резервного копирования. Набор резервного копирования — это общее обозначение, используемое для указания одного или нескольких мест назначения для резервной копии.

В программном обеспечении SQL Server учтено, что резервная копия может оказаться настолько большой, что ее потребуется распределить на нескольких носителях (магнитных лентах или дисках). Необходимость в этом может быть также обусловлена другими причинами. Но если резервная копия распределена на отдельных носителях указанным образом, то в случае восстановления данных все эти носители должны полностью находиться в вашем распоряжении. Иными словами, такая группа носителей представляет собой единый набор. Определение набора резервных копий содержит описание всех мест назначения, используемых при создании какой-то конкретной резервная копии. Кроме того, описание набора резервных копий содержит некоторую информацию о характеристиках рассматриваемой резервной копии. Например, с помощью этого описания можно задать срок хранения резервной копии.

Область окна Destination

Область окна Destination позволяет указать место назначения для резервной копии данных. Именно в этой области можно определить несколько мест назначения, которые применяются в связи с отдельным набором резервных копий. В большинстве инсталляций в качестве места назначения может быть указан каталог, предназначенный для записи файла (который в дальнейшем может быть перезаписан на магнитную ленту), но допускается также возможность указать устройство для резервного копирования, что позволяет вывести резервную копию непосредственно на магнитную ленту или на аналогичное устройство для резервного копирования.

Узел Options

В данном разделе уже описывались опции диалогового окна Back Up Database, доступные после щелчка на узле General, а узел Options этого диалогового окна позволяет определить другие опции. Основная часть этих опций не нуждается в пояснениях, но следует отметить, что именно на странице Options находится область Transaction Log, в которой могут быть заданы режимы ведения журнала транзакций, рассматриваемые выше.

Кнопка Schedule

В диалоговом окне Back Up Database (см. рис. 19.15) предусмотрена также возможность после завершения настройки конфигурации всех параметров создать резервную копию на регулярной основе. Для этой цели предназначена кнопка Schedule, находящаяся в верхней части диалогового окна. После щелчка на этой кнопке открывается диалоговое окно New Job Schedule, описанное ранее в этой главе (см. рис. 19.8). После этого можно определить расписание регулярного запуска подготовленной операции резервного копирования.

Следует отметить, что некоторые дополнительные опции, применяемые при создании резервных копий, могут быть заданы только путем вызова на выполнение команд T-SQL или использования объектной модели SMO из состава средств программирования .NET. Применение этих опций связано с осуществлением довольно сложных действий, поэтому дополнительная информация на эту тему будет приведена в книге *Professional SQL Server 2005 Programming*.

Модели восстановления

В предыдущих разделах рассматривались основные области применения моделей восстановления, а теперь мы можем определить само понятие модели восстановления.

Напомним, что в главе 14, посвященной транзакциям, речь шла о предназначении журнала транзакций. Кратко можно отметить, что журнал транзакций позволяет не только отслеживать ход выполнения транзакций в целях обеспечения отката транзакций и поддержки неразрывности операций, но и восстанавливает данные вплоть до той точки во времени, в которой произошло аварийное завершение работы системы.

Рассмотрим в качестве примера базу данных, эксплуатируемую в банке. Предположим, что последняя полная резервная копия была создана шесть часов тому назад, а с тех пор непрерывно в течение шести часов осуществляются операции зачисления и списания денежных средств. Если в этот момент произойдет аварийное завершение работы системы, то вряд ли кому-то понравится идея восстановить последнюю резервную копию шестичасовой давности и потерять таким образом всю информацию о финансовых операциях, выполненных за истекший период. Это означает, что необходимо обеспечить не только создание резервной копии, но и сохранение всех данных, полученных со времени создания этой копии.

Ведение журнала транзакций предоставляет возможность выполнить так называемую *операцию наката*, которая предусматривает повтор всех транзакций, зафиксированных в базе данных со времени получения последней полной или дифференциальной резервной копии. При условии, что имеются и файлы резервной копии данных, и файлы журнала транзакций, появляется возможность восстановить состояние базы данных вплоть до момента аварийного завершения работы системы.

Модель восстановления определяет, в течение какого времени ведется запись журнала и какая информация в нем фиксируется; предусмотрены три варианта ведения журнала, описанных ниже.

- Full. Этот вариант соответствует своему назначению; при его применении в журнал записывается информация обо всех операциях. При использовании этой модели не должна происходить какая-либо потеря данных в случае аварийного завершения работы системы, при условии, что имеется резервная копия данных, а также доступны все файлы журнала транзакций, созданные со времени получения этой резервной копии. Если какой-либо из журналов отсутствует или содержит искажения в данных, то имеется возможность восстановить все данные, полученные вплоть до времени сохранения последнего неповрежденного журнала. Но следует помнить, что записываются данные обо всех операциях, а это означает, что при наличии большого количества обновлений или вновь введенных данных объем пространства, который потребуется в системе для хранения такого журнала, может оказаться весьма значительным.

- Bulk-Logged. Эта модель восстановления может условно рассматриваться как упрощенный вариант полного восстановления. При использовании опции Bulk-Logged запись в журнал информации об обычных транзакциях осуществляется так же, как и при использовании модели полного восстановления, а информация об операциях массового копирования записывается не в полном объеме. Результатом применения указанной модели восстановления становится то, что в случае аварийного завершения работы системы восстановленная резервная копия будет содержать информацию обо всех изменениях на страницах данных, не относящихся к массовым операциям (таким как массовый импорт данных или создание индекса), поэтому необходимо выполнить все массовые операции повторно. Тем не менее в пользу применения этого метода восстановления свидетельствует то, что в последней версии производительность массовых операций стала намного выше. Но все же, несмотря на высокую производительность, применение операций указанного типа связано с определенным риском возникновения сбоев, и об этом следует помнить, выбирая метод восстановления.
- Simple. При использовании модели восстановления Simple журнал транзакций применяется исключительно для поддержки транзакций в ходе их осуществления. Регулярно проводится очистка журнала транзакций, и в ходе этого из журнала фактически удаляется информация обо всех завершенных транзакциях или транзакциях, в которых был выполнен откат (в действительности осуществляемые при этом действия являются более сложными, но в конечном итоге приводят к получению указанного результата). Таким образом, формируется удобный и насыщенный информацией журнал, который меньше по сравнению с двумя предыдущими вариантами и часто обеспечивает более высокую производительность, но этот журнал является практически непригодным для восстановления, если происходит аварийное завершение работы системы.

Из этого следует вывод, что для базы данных производственного назначения не следует применять какую-либо другую модель восстановления, кроме полной.

Восстановление

Безусловно, задача восстановления является обратной по отношению к задаче резервного копирования. Если резервные копии создавались с соблюдением всех требований, то их можно полностью восстановить, либо для возобновления нормальной работы, либо для создания копии базы данных в другом месте.

После получения резервной копии базы данных сравнительно несложно восстановить эту копию в исходном месте. В своем простейшем варианте операция восстановления осуществляется во многом так же, как и операция резервного копирования. Для этого необходимо перейти к базе данных, которую требуется восстановить, и щелкнуть на ее названии правой кнопкой мыши; затем остается только выбрать команды Tasks⇨Restore, чтобы открыть диалоговое окно Restore Database (рис. 19.16).

Если задача заключается лишь в том, чтобы взять существующую резервную копию и перекрыть с ее помощью базу данных, на основе которой была получена эта резервная копия, то никаких сложностей не возникает. Достаточно щелкнуть на кнопке OK, и восстановление произойдет без каких-либо проблем.

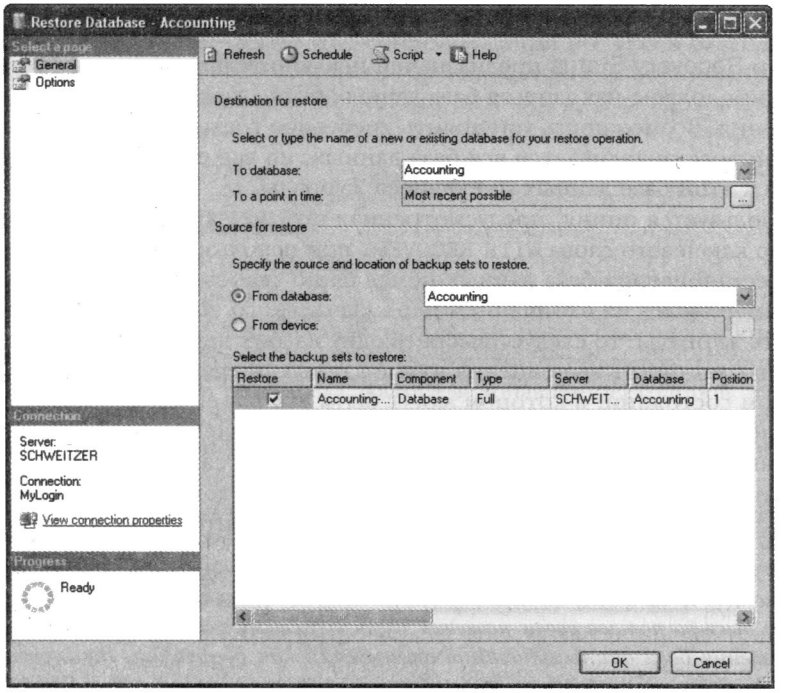


Рис. 19.16. Диалоговое окно Restore Database

Восстановление в другом месте

Задача немного усложняется, если восстановление должно быть выполнено не в том месте, из которого получена резервная копия. Дело в том, что в процессе создания резервной копии регистрируется имя базы данных, из которой извлекается копируемая информация, но, что еще более важно, запоминается обозначение пути (путей) к физическим файлам, применение которых предполагается в восстановленной версии.

Изменить имя базы данных назначения не составляет труда. Это имя можно указать непосредственно в окне, показанном на рис. 19.16. Проблема заключается в том, что смена имени базы данных назначения не позволяет решить более важную задачу — указать другие файлы (файлы `.MDF` и `.LDF`), в которые должна осуществляться запись данных в процессе восстановления. Для этой цели применяется узел Options диалогового окна Restore Database.

Страница, которая откроется после щелчка на узле Options, также в основном содержит параметры, не требующие пояснений, но столбец Restore As заслуживает особого внимания. В этой части диалогового окна можно установить другие значения вместо имени каталога назначения и имени файла применительно к каждому исходному файлу. Это позволяет восстановить несколько копий одной и той же базы данных на одном и том же сервере (возможно, даже в целях проведения экспериментов). Другие варианты состоят в том, что базу данных можно восстановить на другом томе и даже на другом компьютере.

Параметр Recovery Status

Параметр Recovery Status предназначен исключительно для определения состояния, в котором должна находиться база данных после завершения текущей операции восстановления. Возможность установить требуемое состояние становится особенно важной, если восстанавливается вся база данных, но все еще требуется применить в дальнейшем к этой базе данных имеющиеся журналы.

Если используется опция, предусмотренная по умолчанию (которая равносильна применению ключевого слова `WITH RECOVERY` при использовании операторов T-SQL), то после восстановления база данных немедленно становится работоспособной. Но если, например, вслед за окончанием первоначального восстановления необходимо восстановить журналы, то следует выбрать одну из двух других опций. Обе эти опции исключают возможность выполнения в базе данных операций обновления и оставляют ее в таком состоянии, в котором могут быть выполнены дополнительные операции восстановления; различие между этими двумя опциями состоит в том, что при использовании первой из них пользователям разрешается доступ к базе данных в режиме только чтения, а применение второй приводит к тому, что у пользователей попрежнему остается впечатление, что база данных находится в автономном режиме.

Проблема предотвращения доступа пользователей к восстанавливаемой базе данных сложнее, чем может показаться на первый взгляд. Иногда приходится просто удивляться тому, насколько быстро пользователи находят путь в систему после того, как в ходе операции восстановления база данных внезапно отмечается как доступная. Зачастую возникают такие ситуации, что администратор базы данных, подготавливая следующий этап восстановления, не успевает даже открыть базу данных, как обнаруживает, что в ней уже работают пользователи. Если дело обстоит именно так, то обязательно используйте метод восстановления с ключевым словом `NO RECOVERY`. После этого вы сможете приступить к осуществлению этапа восстановления, рассчитанного исключительно на применение опции `WITH RECOVERY`, и полностью перевести базу данных снова в оперативный режим, лишь убедившись в том, что все операции восстановления прошли так, как было задумано.

Сопровождение индексов

Напомним, что в главе 9 речь шла о том, по каким причинам может возникать фрагментация индексов. Со временем из-за фрагментации индексов производительность базы данных значительно снижается, поэтому необходимо иметь возможность устранения такой фрагментации. К счастью, в программном обеспечении SQL Server предусмотрены команды, которые позволяют реорганизовать файлы данных и индексов, чтобы улучшить их внутреннюю структуру. Кроме того, существует возможность применить эти команды в сочетании со средствами планирования заданий, которые были описаны выше в данной книге, и автоматизировать рутинную работу по дефрагментации.

Команды, предназначенные для дефрагментации индексов, в последней версии SQL Server существенно изменились. В предыдущих выпусках применялись средства, известные под общим названием DBCC (Database Consistency Checker; некоторые специалисты также указывают, что эта аббревиатура расшифровывается как Database Console Command). К этим средствам в основном относятся команды `DBCC INDEXDEFRAG`, а также, в меньшей степени, `DBCC DBREINDEX`. В последней версии SQL Server указанные команды заменены новым оператором, `ALTER INDEX`.

В последней версии SQL Server для сопровождения индексов базы данных применяется оператор ALTER INDEX, который гораздо проще в использовании, чем средства DBCC, но при его применении возникают немного другие сложности. В следующем разделе кратко описан синтаксис этого оператора и показано, как запланировать его на выполнение.

Оператор ALTER INDEX

Оператор ALTER INDEX фактически предназначен не для модификации индекса, что, казалось бы, следует из его названия, а для выполнения других действий. Все операторы ALTER, которые рассматривались до сих пор в настоящей книге, предназначены для корректировки определений тех или иных объектов. Например, оператор ALTER применительно к таблицам служил для добавления столбцов, а также ввода в действие или отмены ограничений. С другой стороны, оператор ALTER INDEX имеет иное назначение — он применяется исключительно для сопровождения индексов и не вносит никаких изменений в их структуру. Если требуется изменить определение индекса, то необходимо либо уничтожить его с помощью оператора DROP, а затем создать, применив оператор CREATE, либо воспользоваться оператором CREATE с опцией DROP_EXISTING=ON.

Оператор ALTER INDEX имеет следующий синтаксис:

```
ALTER INDEX { <name of index> | ALL }
   ON <table or view name>
   { REBUILD
      [ [ WITH ( <rebuild index option> [ ,...n ] ) ]
        | [ PARTITION = <partition number>
             [ WITH ( <partition rebuild index option>
                  [ ,...n ] ) ] ] ]
   | DISABLE
   | REORGANIZE
      [ PARTITION = <partition number> ]
      [ WITH ( LOB_COMPACTION = { ON | OFF } ) ]
   | SET ( <set_index_option> [ ,...n ] )
   }
[ ; ]
```

Значительная часть приведенного выше определения синтаксиса относится к сфере деятельности высококвалифицированного администратора базы данных. Как правило, подобные опции используются лишь время от времени, для решения специальных задач. Но некоторые основные элементы синтаксиса этого оператора должны войти в состав планирования регулярного сопровождения. Вначале рассмотрим ряд основных параметров, а затем определим, какие опции должны быть учтены при планировании сопровождения.

Имя индекса

Если необходимо обеспечить сопровождение одного конкретного индекса, то следует указать имя этого индекса с помощью параметра <name of index>, а для выполнения операции сопровождения на каждом индексе, который связан с указанной таблицей, следует ввести ключевое слово ALL.

Имя таблицы или представления

Назначение параметра <table or view name> в основном соответствует его определению — этот параметр позволяет указать имя конкретного объекта (таблицы или представления), на котором должна быть выполнена операция сопровождения.

Следует отметить, что должен быть указан один конкретный объект базы данных (иными словами, не допускается задавать в качестве этого параметра список объектов).

Ключевое слово REBUILD

Применение ключевого слова REBUILD представляет собой самый радикальный подход к сопровождению индексов. После вызова на выполнение оператора ALTER INDEX с этой опцией существующий индекс полностью отбрасывается и воссоздается с нуля. Результатом становится полностью оптимизированный индекс. В этом индексе реконструирована каждая страница, и на уровне листовых узлов, и на всех остальных уровнях, на основе определений, заданных по умолчанию или откорректированных с использованием таких параметров, как коэффициент заполнения.

При использовании этого ключевого слова необходимо соблюдать осторожность. Сразу после вызова на выполнение оператора с опцией REBUILD индекс, с которым вы работаете, становится недоступным до того времени, пока не будет завершена перестройка индекса. Выполнение всех тех запросов, которые основаны на использовании этого индекса, может стать крайне замедленным (замедление может измеряться несколькими порядками величины). Поэтому операция сопровождения индекса с указанной опцией относится к категории таких операций, которые рекомендуется заранее проверять в системе, не находящейся в эксплуатации, чтобы получить представление о том, сколько времени она может потребовать, и только после этого запланировать ее на выполнение в нерабочие часы (рекомендуется также, чтобы кто-то проконтролировал выполнение этой операции для достижения полной уверенности в том, что индекс снова будет введен в действие ко времени возникновения пиковой нагрузки).

Таким образом, при использовании оператора сопровождения индекса с ключевым словом REBUILD приходится учитывать многие важные побочные эффекты, поэтому, как считает автор, применение этого варианта сопровождения должно находиться исключительно в распоряжении администратора базы данных.

Ключевое слово DISABLE

По своему назначению ключевое слово DISABLE (Отменить) полностью соответствует определению. Но следует учитывать, что отмена действия индекса, осуществляемая при использовании этого ключевого слова, приводит к более серьезным последствиям, чем кажется на первый взгляд. Разумеется, было бы неплохо, если бы оператор с этим ключевым словом просто переводил индекс в автономный режим на то время, пока происходит подготовка к дальнейшим действиям, но вместо этого указанный оператор отмечает индекс как непригодный для дальнейшего использования. После отмены действия индекса необходимо выполнить его перестройку (еще раз подчеркиваем, не реорганизацию, а перестройку, с помощью ключевого слова REBUILD), и только после этого индекс снова вводится в действие.

Самому разработчику приходится применять ключевое слово DISABLE крайне редко (вместо его использования гораздо лучше удалить индекс). Ситуации, в которых может потребоваться применение этого ключевого слова, скорее всего, могут возникать в период модернизации программного обеспечения SQL Server или при возникновении каких-то других достаточно редких обстоятельств.

Предупреждение о необходимости соблюдать предельную осторожность относится и к этой опции. Например, если отменяется кластеризованный индекс, который задан на таблице, то по существу налагается запрет на использование таблиц. Данные в таблице остаются незатронутыми, но становятся недостижимыми с применением любых индексов до завершения перестройки кластеризованного индекса, поскольку от кластеризованного индекса зависят все прочие индексы.

Ключевое слово REORGANIZE

С точки зрения разработчика, наиболее подходящим является оператор сопровождения индекса с ключевым словом REORGANIZE. В результате реорганизации индекса достигается немного менее полная оптимизация по сравнению с полной перестройкой, но преимуществом реорганизации является то, что она происходит в оперативном режиме (пользователи могут по-прежнему работать с индексом).

Оптимизация является не такой полной, как при перестройке, поскольку реорганизация выполняется только на уровне листовых узлов индекса; уровни индекса, не относящиеся к уровню листовых узлов, остаются незатронутыми. Это означает, что при этом достигается степень оптимизации, немного меньшая по сравнению с возможной, но для подавляющего большинства индексов характерно наличие фрагментации в основном на самом низком уровне иерархической структуру индекса. Тем не менее в некоторых случаях достигаемая степень оптимизации может оказаться недостаточной, поэтому придется воспользоваться другим методом сопровождения индекса.

Учитывая то, что оператор сопровождения индекса с ключевым словом REORGANIZE оказывает наименьшее отрицательное влияние на работу пользователей, обычно именно это инструментальное средство становится частью плана регулярного сопровождения индексов. Поэтому ниже описано, как применить на практике оператор реорганизации индекса.

Практическое занятие Реорганизация индекса

В качестве примера рассмотрим, как осуществляется реорганизация индекса на таблице базы данных AdventureWorks. Превосходным примером таблицы, в которую с наибольшей вероятностью со временем вставляется большое количество строк, после чего эти строки удаляются в связи с тем, что истекает интервал времени, в течение которого необходимо хранить информацию о транзакциях, является таблица Production.TransactionHistory. В данном случае для реорганизации всех индексов, заданных на этой таблице, применяется следующий простой оператор:

```
ALTER INDEX ALL
ON Production.TransactionHistory
REORGANIZE;
```

После выполнения этого оператора из СУБД не поступает какая-либо содержательная информация; передается лишь простое сообщение об успешном завершении: "Command(s) completed successfully".

Описание полученных результатов

При выполнении оператора ALTER INDEX программное обеспечение СУБД определяет, что вместо конкретного имени индекса задано ключевое слово ALL. В связи с этим осуществляется поиск всех индексов, заданных на таблице Production.TransactionHistory (но из полученного списка индексов исключаются индексы, которые отменены, поскольку операция реорганизации на такие индексы не распространяется). Затем незаметно для пользователя происходит обработка в цикле всех индексов, предназначенных для реорганизации. Для этого выбирается последовательно один индекс за другим и выполняется его реорганизация только на уровне листовых узлов (включая реорганизацию действительных данных, поскольку эта операция распространяется также на кластеризованный индекс таблицы).

Архивирование данных

Задача архивирования данных является очень сложной. Достаточно сказать, что, наверное, каждый специалист по конструированию баз данных предлагает собственный способ архивирования данных. К тому же многое зависит от самого приложения. Например, если создается база данных OLAP, предназначенная для использования в сочетании со службами Analysis Services, то приходится учитывать, насколько далеко в прошлое могут быть обращены аналитические запросы. Но независимо от используемого метода определения допустимого срока хранения данных, рано или поздно наступает такой день, когда приходится сталкиваться с проблемой удаления из работающей системы устаревших данных, из-за большого объема которых производительность неизбежно снижается.

Весьма значительное разнообразие способов архивирования обусловлено также тем, что каждая база данных имеет свои отличительные особенности. Но ключом к решению задачи архивирования является прогнозирование потребностей в архивировании еще до того, как будет создана сама база данных. Необходимо учитывать, что строки, переносимые в архив, должны быть удалены, а это может привести к нарушению ограничений ссылочной целостности и (или) появлению висячих строк. Это означает, что еще при проектировании базы данных необходимо продумать алгоритмы удаления или перемещения строк во время архивирования. Ниже приведены некоторые рекомендации по разработке сценариев архивирования.

- Если данные уже хранятся в базе данных OLAP, то, по-видимому, запись этих данных в архив в связи с удалением из базы данных не требуется. Но этот вопрос должен быть решен на уровне руководства компании.
- Определите, как часто архивные данные используются в действительности и насколько дорого обходится их хранение. Люди от природы склонны накапливать у себя всякое старье. Иначе говоря, многие очень неохотно избавляются от ненужных вещей; к этому относятся и данные. Но иногда в основе стремления сохранить давно устаревшие данные лежат такие опасения, что уничтожение этих данных может войти в противоречие с требованиями закона. В таком случае вполне возможно предусмотреть хранение копий никогда не используемых или редко используемых данных на магнитной ленте (для хранения архивных данных можно также порекомендовать использование нескольких резервных копий) и сокращение объема данных, доступных в оперативном режиме. В результате этого пользователи сразу же обнаружат, что повысилась производительность системы и стало легче работать.
- Не оставляйте висячие строки. Если удаление архивированных данных осуществляется в условиях применения ограничений ссылочной целостности, то вероятность появления висячих строк невелика, но если такие ограничения не применяются, появление висячих строк становится неизбежным. Такая ситуация может привести к возникновению серьезных ошибок в системе.
- Выясните, не потребует ли выполнение программы архивирования много времени. Если такая программа выполняется слишком долго и затрагивает большое количество строк, то могут возникнуть проблемы при обеспечении параллельного доступа к данным, к которым обращаются пользователи, работающие в оперативном режиме. Поэтому операции архивирования необходимо планировать на выполнение в такое время, когда система не используется.
- Заблаговременно и тщательно проверяйте все процедуры архивирования.

Резюме

Настоящая глава не только предоставляет возможность воспользоваться важными рекомендациями, но и заставляет о многом задуматься. В частности, для многих разработчиков весьма характерно стремление рассматривать задачи администрирования базы данных как не относящиеся к сфере их компетенции. Но этот подход — неправильный!

Несколько лет тому назад автору довелось столкнуться с одной ситуацией, которая стала замечательным примером того, что разработчики должны брать на себя ответственность за все происходящее в связи с эксплуатацией созданного ими приложения. Для некоммерческой группы, которая работает на северо-западе Соединенных Штатов, по специальному заданию была разработана замечательная система. Но примерно через восемь месяцев после начала эксплуатации этой системы в компанию, разработавшую программное обеспечение, поступил тревожный звонок. После обсуждения всех обстоятельств происшествия было определено, что в файлах базы данных возникло какое-то искажение информации, поэтому специалисты компании порекомендовали заказчику восстановить базу данных из резервной копии. Но в ответ прозвучал недоуменный вопрос: "А что такое резервная копия?" Рассматриваемая в данном примере компания-разработчик упустила нечто очень важное. Ее специалисты знали, что перед ними — неопытный заказчик, который не собирается привлекать к администрированию базы данных квалифицированный персонал. Кто же тогда должен был сообщить заказчику, что нужно создавать резервные копии, и помочь ему организовать такую работу, если об этом не подумали сами разработчики? Я счастлив сообщить, что рассматриваемая здесь компания извлекла урок из полученного ею опыта, что не мешает также сделать всем прочим разработчикам.

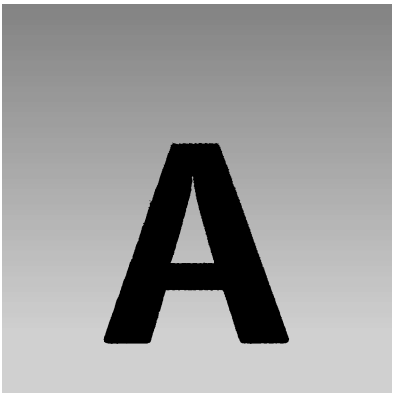
О проблемах администрирования базы данных необходимо задумываться еще на этапе проектирования, и тем более не следует забывать об этом, подготавливая планы внедрения системы в эксплуатацию. Если для упрощения администрирования системы сделано все возможное, то функционирование самой системы протекает намного более успешно, а это, в свою очередь, приносит значительное вознаграждение для разработчика.

Упражнения

19.1. Возьмите за основу команду, приведенную в примере реорганизации таблицы `Production.TransactionHistory` на с. 731, и запланируйте ее на выполнение в качестве задания выходного дня на 2 часа утра в воскресенье.

19.2. Выполните полное резервное копирование базы данных `Pubs`. Сохраните резервную копию под именем `C:\MyBackup.bak` (если в разделе `C:` отсутствует достаточный объем дискового пространства, разрешается выбрать другой раздел жесткого диска).

19.3. Восстановите резервную копию, которая была создана в результате выполнения упражнения 19.2, чтобы создать такую же базу данных, но с новым именем, `NewPubs`.



Ответы к упражнениям

Глава 3

3.1. Напишите запрос, который выводит все столбцы и все строки из таблицы authors базы данных pubs.

Определите pubs в качестве текущей базы данных и выполните запрос:

```
SELECT * FROM authors
```

3.2. Измените запрос, рассматриваемый в упражнении 3.1, таким образом, чтобы в полученных результатах остались сведения только об авторах из штата Юта (Utah). (Подсказка. Их должно быть 2.)

```
SELECT * FROM authors WHERE state = 'UT'
```

3.3. Добавьте новую строку в таблицу authors базы данных pubs.

```
INSERT INTO authors VALUES ('999-99-9999', 'Spade', 'Sam', '333 555-1212',
'123 Main St.', 'Seattle', 'WA', '99999', 1)
```

3.4. Удалите только что добавленную строку.

В соответствии с примером решения упражнения 3.3 можно выполнить следующее:

```
DELETE authors WHERE au_id = '999-99-9999')
```

Глава 4

4.1. Напишите запрос к базе данных Northwind, который возвращает один столбец SupplierName и содержит имя поставщика товара Chai.

```
SELECT CompanyName AS SupplierName FROM Products p JOIN Suppliers s ON
p.SupplierID = s.SupplierID WHERE p.ProductName = 'Chai')
```

4.2. Напишите запрос с конструкцией JOIN, чтобы получить список с указанием каждой территории в базе данных Northwind, для обслуживания которой не назначен служащий. (Подсказка. Используйте внешнее соединение.)

```
SELECT TerritoryDescription FROM Territories t LEFT JOIN
EmployeeTerritories et ON t.TerritoryID = et.TerritoryID WHERE
et.TerritoryID IS NULL)
```

Глава 5

5.1. Создайте сценарии SQL для таблиц Customers и Employees, используя средства формирования сценариев программы Management Studio.

```
   USE [Northwind]
GO
/****** Object:  Table [dbo].[Customers]    Script Date: 08/26/2005
07:12:46 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Customers](
    [CustomerID] [nchar](5) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [CompanyName] [nvarchar](40) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [ContactName] [nvarchar](30) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [ContactTitle] [nvarchar](30) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Address] [nvarchar](60) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [City] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Region] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [PostalCode] [nvarchar](10) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Country] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Phone] [nvarchar](24) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Fax] [nvarchar](24) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
 CONSTRAINT [PK_Customers] PRIMARY KEY CLUSTERED
(
    [CustomerID] ASC
) ON [PRIMARY]
) ON [PRIMARY]
GO
SET ANSI_NULLS OFF
GO
SET QUOTED_IDENTIFIER OFF
USE [Northwind]
GO
/****** Object:  Table [dbo].[Employees]    Script Date: 08/26/2005
07:13:21 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Employees](
    [EmployeeID] [int] IDENTITY(1,1) NOT NULL,
```

```
    [LastName] [nvarchar](20) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [FirstName] [nvarchar](10) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [Title] [nvarchar](30) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [TitleOfCourtesy] [nvarchar](25) COLLATE
SQL_Latin1_General_CP1_CI_AS NULL,
    [BirthDate] [datetime] NULL,
    [HireDate] [datetime] NULL,
    [Address] [nvarchar](60) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [City] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Region] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [PostalCode] [nvarchar](10) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Country] [nvarchar](15) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [HomePhone] [nvarchar](24) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Extension] [nvarchar](4) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [Photo] [image] NULL,
    [Notes] [ntext] COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
    [ReportsTo] [int] NULL,
    [PhotoPath] [nvarchar](255) COLLATE SQL_Latin1_General_CP1_CI_AS NULL,
 CONSTRAINT [PK_Employees] PRIMARY KEY CLUSTERED
(
    [EmployeeID] ASC
) ON [PRIMARY]
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
SET ANSI_NULLS OFF
GO
SET QUOTED_IDENTIFIER OFF
GO
USE [Northwind]
GO
ALTER TABLE [dbo].[Employees]  WITH NOCHECK ADD  CONSTRAINT
[FK_Employees_Employees] FOREIGN KEY(    [ReportsTo])
REFERENCES [dbo].[Employees] (    [EmployeeID])
GO
ALTER TABLE [dbo].[Employees] CHECK CONSTRAINT [FK_Employees_Employees]
GO
ALTER TABLE [dbo].[Employees]  WITH NOCHECK ADD  CONSTRAINT
[CK_Birthdate] CHECK  (([BirthDate]<getdate()))
GO
ALTER TABLE [dbo].[Employees] CHECK CONSTRAINT [CK_Birthdate]
```

5.2. Подготовьте сценарий создания базы данных MyDB без использования программы Management Studio. Установите начальный размер базы данных 17MB и начальный размер журнала 5MB; увеличение размеров базы данных и журнала должно происходить с шагом 5MB.

Один из нескольких возможных ответов приведен ниже.

```
CREATE DATABASE MyDB ON PRIMARY( NAME='MyDB', FILENAME='C:\Myfile.mdf',
SIZE = 17MB, FILEGROWTH = 5MB ) LOG ON ( NAME='MyDBLog',
FILENAME='C:\MyLog.ldf', SIZE = 5MB, FILEGROWTH = 5MB )
```

5.3. Создайте таблицу Foo с единственным символьным полем переменной длины Col1; установите ограниченный размер столбца Col1, равный 50 символам.

```
CREATE TABLE Foo (Col1   varchar(50))
```

Глава 7

7.1. Напишите запрос, который возвращает сведения о том, какова дата приема на работу каждого из служащих компании Northwind, в формате MM/DD/YY.

```
SELECT CONVERT(varchar(12), HireDate, 1) FROM Employees
```

7.2. Напишите отдельные запросы с использованием соединения, подзапроса, а затем оператора EXISTS, чтобы составить список всех заказчиков компании Northwind, которые не разместили в ней свои заказы.

```
   SELECT CompanyName
FROM Customers c
LEFT JOIN Orders o
   ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL
SELECT CompanyName
FROM Customers C
WHERE c.CustomerID NOT IN (SELECT CustomerID FROM Orders o)
SELECT CompanyName
FROM Customers c
WHERE NOT EXISTS (SELECT CustomerID FROM Orders o WHERE o.CustomerID =
c.CustomerID)
```

7.3. Покажите пять самых последних заказов, сделанных заказчиком, который потратил больше 25 тыс. долл. на приобретение товаров в компании Northwind.

```
   SELECT TOP 5 *
FROM Orders oo
WHERE CustomerID IN
(
   SELECT c.CustomerID
   FROM Customers c
   JOIN Orders o
      ON c.CustomerID = o.CustomerID
   JOIN [Order Details] od
      ON o.OrderID = od.OrderID
   GROUP BY c.CustomerID
   HAVING SUM(UnitPrice * (1-Discount) * Quantity) > 25000
)
ORDER BY OrderDate
```

Этот запрос можно было бы также написать с использованием соединения и производной таблицы.

Глава 8

8.1. Нормализуйте данные, приведенные в табл. А.1, с преобразованием в третью нормальную форму.

Таблица А.1. Исходные данные для решения упражнения 8.1

Patient	SSN	Physician	Hospital	Treatment	AdmitDate	ReleaseDate
Sam Spade	555-55-5555	Albert Schweitzer	Mayo Clinic	Labotomy	10/01/2005	11/07/2005
Sally Nally	333-33-3333	Albert Schweitzer	NULL	Cortezone Injection	10/10/2005	10/10/2005
Peter Piper	222-22-2222	Mo Betta	Mustard Clinic	Pickle Extraction	11/07/2005	11/07/2005
Nicki Doohickey	123-45-6789	Sheeze Sheila	Mustard Clinic	Cortezone Injection	11/07/2005	11/07/2005

Решение данного упражнения приведено в табл. А.2–А.6.

Таблица А.2. Таблица Patients

PatientID	PatientName	SSN
1	Sam Spade	555-55-5555
2	Sally Nally	333-33-3333
3	Peter Piper	222-22-2222
4	Nicki Doohickey	123-45-6789

Таблица А.3. Таблица Physicians

PhysicianID	PhysicianName	Address
1	Albert Schweitzer	1234 anywhere
2	Mo Betta	567 Main
3	Sheeze Sheila	89 1st

Таблица А.4. Таблица Hospitals

HospitalID	HospitalName	Address
1	Mayo Clinic	1234 Anywhere
2	Mustard Clinic	55 French's Avenue

Таблица А.5. Таблица Treatments

TreatmentID	TreatmentDescription
1	Labotomy
2	Cortezone Injection
3	Pickle Extraction
4	Cortezone Injection

Таблица А.6. Таблица Visits

VisitID	PatientID	PhysicianID	HospitalID	TreatmentID	AdmitDate	ReleaseDate
1	1	1	1	1	10/01/2005	11/07/2005
2	2	1	NULL	2	10/10/2005	10/10/2005
3	3	2	2	3	11/07/2005	11/07/2005
4	4	3	2	4	11/07/2005	11/07/2005

Глава 9

9.1. Назовите по меньшей мере два способа определения того, какие индексы заданы на таблице HumanResources.Employee базы данных AdventureWorks.

Можно указать по меньшей мере два описанных ниже способа определения того, какие индексы заданы на таблице HumanResources.Employee базы данных AdventureWorks.

а) Перейти в окне Management Studio к узлу Indexes под именем таблицы HumanResources.Employee в списке Tables, относящемся к базе данных AdventureWorks, и развернуть этот узел. Раскроется список, включающий все индексы, на котором можно дважды щелкать мышью для получения дополнительной информации.

б) Воспользоваться командой sp_help 'HumanResources.Employee' или sp_helpindex 'HumanResources.Employee'. (Это позволяет достичь одинаковых результатов, если не считать того, что вывод хранимой процедуры sp_help не только включает все данные, формируемые с помощью процедуры sp_helpindex, но и содержит дополнительные сведения.)

9.2. Создать некластеризованный индекс на столбце ModifiedDate таблицы Production.ProductModel базы данных AdventureWorks.

```
CREATE NONCLUSTERED INDEX ieProductModel ON Production.ProductModel(ModifiedDate)
```

9.3. Удалить индекс, созданный при выполнении упражнения 9.2.

```
DROP INDEX Production.ProductModel.ieProductModel
```

Глава 10

10.1. Создайте в базе данных Northwind представление Managers, с помощью которого можно получить данные только о тех служащих, которые выполняют обязанности руководителей по отношению к другим служащим.

```
CREATE VIEW Managers
AS
SELECT * FROM Employees o
WHERE EXISTS (SELECT 'true' FROM Employees i WHERE i.ReportsTo = o.EmployeeID)
```

10.2. Модифицируйте представление, созданное по условиям упражнения 10.1, так, чтобы оно стало зашифрованным.

```
ALTER VIEW Managers
WITH ENCRYPTION
AS
SELECT * FROM Employees o
WHERE EXISTS (SELECT 'true' FROM Employees i WHERE i.ReportsTo = o.EmployeeID)
```

10.3. Повторно создайте и индексируйте существующее представление "Products by Category" базы данных Northwind на основе столбцов CategoryName и ProductName.

```
   SET ANSI_NULLS ON
GO
```

```
SET QUOTED_IDENTIFIER ON
GO
ALTER view [dbo].[Products by Category]
WITH SCHEMABINDING
AS
SELECT c.CategoryName, p.ProductName, p.QuantityPerUnit, p.UnitsInStock,
p.Discontinued
FROM dbo.Categories c
JOIN dbo.Products p
   ON c.CategoryID = p.CategoryID
WHERE p.Discontinued <> 1
GO
CREATE UNIQUE CLUSTERED INDEX ivProductsByCategory
ON [Products by Category](CategoryName, ProductName)
SET ANSI_NULLS OFF
GO
SET QUOTED_IDENTIFIER OFF
GO
```

Глава 11

11.1. Напишите простой сценарий, в котором создаются две целочисленные переменные (с именами `Var1` и `Var2`), в них помещаются соответственно значения 2 и 4, а затем осуществляется суммирование и вывод суммы значений этих двух переменных.

```
DECLARE @Var1 int
DECLARE @Var2 int
SET @Var1 = 2
SET @Var2 = 4
SELECT @Var1 + @Var2
```

11.2. Создайте переменную `MinOrder` и присвойте ей значение, равное минимальной стоимости товарной позиции в заказе заказчика с идентификатором `CustomerNo`, равным `'ALFKI'`, хранящемся в базе данных `Northwind`. (Предостережение. В данном случае речь идет об обработке денежных сумм, поэтому не следует использовать для переменной `MinOrder` тип данных `int`.) После этого выведите окончательное значение `MinOrder`.

```
DECLARE @MinOrder             money
SELECT @MinOrder = MIN(Quantity * UnitPrice * (1-Discount))
FROM [Order Details] od
JOIN Orders o
       ON od.OrderID = o.OrderID
WHERE CustomerID = 'ALFKI'
SELECT @MinOrder
```

11.3. Выведите результаты выполнения запроса `SELECT COUNT(*) FROM Customers` в окно терминала с использованием утилиты SQLCMD.

```
C:\SQLCMD -UMyLogin -PMyPassword -q"SELECT COUNT(*) FROM Customers"
```

Глава 12

12.1. Напишите простую хранимую процедуру, которая возвращает требуемую строку с информацией о заказчике из базы данных Northwind после получения параметра CustomerID.

```
CREATE spListCustomer @CustomerID
SELECT * FROM Customers
WHERE CustomerID = @CustomerID
```

12.2. Напишите хранимую процедуру, которая принимает данные об идентификаторе территории (TerritoryID), описании территории (TerritoryDescription) и идентификаторе региона (RegionID) и вставляет их в виде новой строки в таблицу Territories базы данных Northwind.

```
CREATE PROC spAddTerritory @TerritoryID nvarchar(20),
@TerritoryDescirption nchar(50), @RegionID int
AS
   INSERT INTO Territories (TerritoryID, TerritoryDescription, RegionID)
     VALUES (@TerritoryID, @TerritoryDescirption, @RegionID)
```

12.3. Модифицируйте хранимую процедуру, созданную при выполнении упр. 12.2, чтобы обеспечить выполнение в ней предварительной проверки наличия внешнего ключа (RegionID) перед осуществлением попытки вставки. Если соответствующее значение RegionID не существует, активизируйте сообщение об ошибке "RegionID is not valid. Please check your RegionID and try again" (Недопустимое значение RegionID. Проверьте значение RegionID и повторите ввод).

```
ALTER PROC spAddTerritory @TerritoryID nvarchar(20),
@TerritoryDescirption
nchar(50), @RegionID int
AS
BEGIN
   DECLARE @Count INT
   SELECT @Count = COUNT(*)
     FROM Region
     WHERE RegionID = @RegionID
   IF @Count < 1
      RAISERROR ('RegionID is not valid. Please check your RegionID and try
again.', 11, 1)
   ELSE
     INSERT INTO Territories (TerritoryID, TerritoryDescription, RegionID)
     VALUES (@TerritoryID, @TerritoryDescirption, @RegionID)
END
```

12.4. Модифицируйте процедуру, созданную при выполнении упр. 12.2, чтобы обеспечить в ней обработку исключительной ситуации, возникшей после обнаружения того, что значение RegionID не существует. Организуйте перехват всех прочих ошибок и предусмотрите для них общее сообщение об ошибке: "An unhandled exception has occurred. Contact your system administrator" (Возникла необработанная исключительная ситуация. Обратитесь к системному администратору).

```
ALTER PROC spAddTerritory @TerritoryID nvarchar(20),
@TerritoryDescirption nchar(50), @RegionID int
AS
BEGIN
   BEGIN TRY
     INSERT INTO Territories (TerritoryID, TerritoryDescription, RegionID)
     VALUES (@TerritoryID, @TerritoryDescirption, @RegionID)
   END TRY
   BEGIN CATCH
     IF ERROR_NUMBER() = 547
       RAISERROR ('RegionID is not valid. Please check your RegionID and try
again.', 11, 1)
     ELSE
       RAISERROR ('An unhandled exception has occurred. Contact your system
administrator', 11, 1)
   END CATCH
END
```

Глава 13

13.1. Повторно реализуйте сценарий `spTriangular`, рассматриваемый в главе 12, но на этот раз в виде функции, а не хранимой процедуры.

```
CREATE FUNCTION dbo.fnTriangular
(@ValueIn AS int)
RETURNS int
AS
BEGIN
 DECLARE @ValueOut  int
 DECLARE @InWorking int
 IF @ValueIn != 1
 BEGIN
             SELECT @InWorking = @ValueIn - 1
             SELECT @ValueOut = @ValueIn + dbo.fnTriangular(@InWorking)
 END
 ELSE
 BEGIN
             SELECT @ValueOut = 1
 END
 RETURN (@ValueOut)
END
```

Глава 19

19.1. Возьмите за основу команду, приведенную в примере реорганизации таблицы `Production.TransactionHistory` на с. 731, и запланируйте ее на выполнение в качестве задания выходного дня на 2 часа утра в воскресенье.

Чтобы запланировать указанную команду на выполнение, необходимо осуществить описанные ниже действия.

а) Перейдите к узлу SQL Server Agent, щелкните правой кнопкой мыши на названии каталога Jobs и выберите элемент New Job.

б) Присвойте имя новому заданию и, по желанию, в качестве категории задайте Database Maintenance; после этого выберите страницу Steps.

в) Выберите New и присвойте имя новому шагу. Оставьте значение T-SQL в качестве опции типа, а в раскрывающемся списке баз данных укажите базу данных `AdventureWorks`.

г) В окне ввода команд введите оператор реорганизации индекса (оператор ALTER INDEX, применявшийся для сопровождения индексов этой таблицы, который приведен на с. 731) и щелкните на кнопке OK.

д) Выберите страницу Schedules и щелкните на кнопке New.

е) Присвойте имя новому расписанию, выберите Recurring в качестве обозначения типа расписания, выберите Weekly в поле Occurs и установите флажок Sunday. В окне Daily frequency щелкните на переключателе Occurs once at, установите в окне счетчика значение 02:00.00 AM и щелкните на кнопке OK.

ж) Щелкните на кнопке OK основного диалогового окна, после чего выполнение данного упражнения завершается.

19.2. Выполните полное резервное копирование базы данных Pubs. Сохраните резервную копию под именем C:\MyBackup.bak (если в разделе C: отсутствует достаточный объем дискового пространства, разрешается выбрать другой раздел жесткого диска).

Чтобы получить резервную копию, необходимо выполнить описанные ниже действия.

а) Перейдите к узлу базы данных Pubs, щелкните на нем правой кнопкой мыши и выберите команду Tasks⇨Backup.

б) Щелкните на кнопке Add области Destination в нижней части диалогового окна и введите C:\MyBackup.bak во всплывающем окне.

в) Выберите заданное по умолчанию местонахождение данных (отличное от того, которое было только что указано) и щелкните на кнопке Remove. Щелкните на кнопке OK.

19.3. Восстановите резервную копию, которую была создана в результате выполнения упражнения 19.2, чтобы создать такую же базу данных, но с новым именем, NewPubs.

Для восстановления резервной копии необходимо выполнить описанные ниже действия.

а) Щелкните правой кнопкой мыши на узле Databases и выберите команду Restore Database.

б) Введите NewPubs типа в поле To Database.

в) Выберите From Device и щелкните на кнопке со знаком многоточия (...).

г) Щелкните на кнопке Add в диалоговом окне Specify Backup и перейдите к обозначению C:\MyBackup.bak; щелкните на кнопке OK.

д) Еще раз щелкните на кнопке OK в диалоговом окне Specify Backup, чтобы возвратиться к основному диалоговому окну Restore.

е) Щелкните на обозначении резервной копии, которое было только что сформировано в области Select backups to restore, а затем выберите страницу Options диалогового окна (в верхней левой части диалогового окна Restore Database).

ж) Задайте вместо имени файла pubs.mdf значение newpubs.mdf, а вместо pubs_log.ldf — значение newpubs_log.ldf.

з) Щелкните на кнопке OK и ожидайте завершения восстановления.

Б

Системные переменные и функции

Программное обеспечение SQL Server поддерживает большое количество системных переменных и функций. Некоторые из них используются довольно часто и в основном не требуют подробных пояснений, а другие применяются реже и должны быть описаны более полно.

В настоящем приложении предпринята попытка дать точное и краткое описание большинства системных переменных и функций SQL Server.

Следует отметить, что в предыдущих выпусках многие системные переменные часто именовались глобальными переменными. Такое применение термина "глобальная переменная" является неправильным, поэтому корпорацией Microsoft при подготовке нескольких последних выпусков прилагаются значительные усилия по внесению исправлений в документацию, необходимых для его замены более подходящим термином "системная переменная". В данном разделе упминание об устаревшей терминологии приведено для того, чтобы читатель мог понять, что подразумевается под словосочетанием "глобальная переменная", если он встретит его во время работы с программным обеспечением SQL Server.

Системные переменные и функции T-SQL, которые предоставляются в программном обеспечении SQL Server 2005, подразделяются на 11 категорий, описанных ниже.

- Системные переменные (которые прежде иногда именовались глобальными переменными).
- Агрегирующие функции.
- Функции для работы с курсорами.
- Функции для работы со значениями даты и времени.
- Математические функции.
- Функции для работы с метаданными.

- Функции для работы с наборами строк.
- Функции защиты.
- Строковые функции.
- Системные функции.
- Функции для работы с текстом и изображениями.

Системные переменные (которые прежде иногда именовались глобальными переменными)

Системная переменная @@CONNECTIONS

Содержит информацию о количестве попыток установить соединение, предпринятых со времени последнего запуска СУБД SQL Server.

Эта системная переменная предоставляет сведения об общем количестве всех попыток создать соединение, сделанных с тех пор, как в последний раз был выполнен запуск СУБД SQL Server. Таким образом, наиболее важная особенность этой переменной состоит в том, что она позволяет узнать, сколько было сделано попыток, а не фактически установлено соединений, кроме того, предоставляет информацию, касающуюся соединений, а не пользователей.

Значение системной переменной `@@CONNECTIONS` представляет собой счетчик, значение которого увеличивается после каждой предпринятой попытки установить соединение, независимо от того, было ли соединение установлено успешно или нет. Единственный нюанс, связанный с применением этой переменной, заключается в том, что попытка установить соединение должна быть обнаружена сервером базы данных. Если в ходе создания соединения один из этапов этого процесса окончился неудачей, скажем, из-за наличия неправильной версии библиотеки `NetLib` или в результате возникновения какой-то другой сетевой проблемы, то СУБД SQL Server не получит уведомление о том, что значение `@@CONNECTIONS` должно быть увеличено. Следует рассчитывать на получение информации только о таких попытках соединения, которые обнаружены сервером. При этом не имеет значения, окончилась ли эта попытка успешно или неудачно.

Важно также учитывать, что речь идет о попытках установить соединение, а не войти в систему. В зависимости от приложения может быть предусмотрено создание нескольких соединений с сервером, но получение от пользователя регистрационной информации, по-видимому, должно осуществляться только единожды. И действительно, такая организация работы предусмотрена даже в программе Query Analyzer. После щелчка на команде меню, для выполнения которой требуется открыть новое окно, программа Query Analyzer автоматически создает еще одно соединение на основе той информации о пользователе, с помощью которой был осуществлен вход в систему.

Для получения значения системной переменной `@@CONNECTIONS`, как и многих других системных переменных, удобно также использовать системную хранимую процедуру `sp_monitor`. Эта процедура, вызов которой осуществляется без параметров, предоставляет информацию о количестве соединений, загруженности процессора, а также о многих других показателях, вплоть до общего количества операций записи, выполненных в СУБД SQL Server.

Системная переменная @@CPU_BUSY

Системная переменная @@CPU_BUSY позволяет получить информацию о том, какова продолжительность времени, измеряемая в миллисекундах с момента последнего запуска СУБД SQL Server, в течение которого процессор был занят выполнением работы. Точность полученного значения зависит от степени дискретизации показаний системного таймера, которая может быть установлена по-разному, в зависимости от конкретной системы.

Системная переменная @@CPU_BUSY представляет собой еще один пример переменной, значение которой измеряется с момента запуска сервера. Из этого следует, что всегда можно рассчитывать на неизменное возрастание этого значения в ходе эксплуатации приложения. Таким образом, контроль над изменением значения системной переменной @@CPU_BUSY позволяет, в частности, определить, какая процентная доля процессорного времени расходуется в процессе эксплуатации СУБД SQL Server. Тем не менее более реалистичный подход к решению указанной задачи состоит, скорее, в использовании программы Performance Monitor. В целом можно сделать вывод, что системная переменная @@CPU_BUSY относится к числу системных переменных такого типа, которые, хотя и могут, на первый взгляд, рассматриваться как основа для создания поистине великолепных разработок, в действительности почти не находят практического применения в большинстве приложений.

Системная переменная @@CURSOR_ROWS

Системная переменная @@CURSOR_ROWS позволяет получить информацию о том, какое количество строк находится в настоящее время в последнем наборе данных курсора, открытом в текущем соединении. Следует подчеркнуть, что эти данные относятся к курсорам, а не к временным таблицам.

Необходимо отметить, что значение этой переменной переустанавливается каждый раз после открытия нового курсора. Если требуется открыть больше одного курсора одновременно и иметь в своем распоряжении информацию о том, сколько строк находится в наборе данных каждого курсора, то необходимо перед открытием каждого следующего курсора переносить значение системной переменной @@CURSOR_ROWS в какую-то промежуточную переменную.

Безусловно, системная переменная @@CURSOR_ROWS может использоваться для установки счетчика для управления циклом WHILE, применяемого для обработки набора данных курсора, но автор настоятельно рекомендует не придерживаться такой практики. Дело в том, что значение, содержащееся в переменной @@CURSOR_ROWS, может изменяться в зависимости от типа курсора и от того, происходит ли заполнение набора данных курсора программным обеспечением SQL Server асинхронно. Способ управления циклом с использованием системной переменной @@FETCH_STATUS является гораздо более надежным; к тому же он является не более сложным.

Если значение, содержащееся в системной переменной @@CURSOR_ROWS, представляет собой отрицательное число больше чем -1, то, по-видимому, применяется асинхронный курсор, а отрицательное число обозначает количество строк, созданных до сих пор в курсоре. С другой стороны, если полученное значение равно -1, то курсор представляет собой динамический курсор, в котором количество строк постоянно изменяется. Значение, равное 0, свидетельствует о том, что либо курсор не открыт, либо последний открытый курсор уже закрыт. Наконец, любое положительное число указывает количество строк в курсоре.

Чтобы создать асинхронный курсор, установите с помощью процедуры `sp_configure` *значение параметра* `cursor_threshold` *больше* `0`. *В таком случае после превышения указанного параметра системной переменной* `@@CURSOR_ROWS` *присваивается значение количества строк в наборе данных курсора, а остальные строки помещаются в курсор асинхронно.*

Системная переменная @@DATEFIRST

Системная переменная `@@DATEFIRST` содержит числовое значение, которое соответствует дню недели, рассматриваемому в системе как первый день недели.

По умолчанию в Соединенных Штатах применяется значение `7`, которое соответствует воскресенью. Применяемые числовые обозначения дней недели приведены ниже.

- `1`. Понедельник (рассматривается как первый день недели в большинстве стран мира).
- `2`. Вторник.
- `3`. Среда.
- `4`. Четверг.
- `5`. Пятница.
- `6`. Суббота.
- `7`. Воскресенье.

Системная переменная `@@DATEFIRST` может применяться для решения проблем локализации, связанных с учетом национальных традиций того государства, в котором применяется система. С ее помощью может быть, например, правильно выбрана компоновка календаря или упорядочена другая информация, в которой учитываются дни недели.

Для изменения параметра настройки конфигурации, которой обозначается первый день недели, используется функция `SET` *с параметром* `DATEFIRST`.

Системная переменная @@DBTS

Системная переменная `@@DBTS` содержит информацию о последней используемой временной отметке для текущей базы данных.

На первый взгляд кажется, что системная переменная `@@DBTS` по своему назначению почти не отличается от системной переменной `@@IDENTITY`, которая содержит последнее идентификационное значение, установленное системой. Но переменная `@@DBTS` позволяет определить последнюю временную отметку, а не последнее идентификационное значение. При использовании переменной `@@DBTS` необходимо учитывать перечисленные ниже особенности.

- Значение этой переменной изменяется в результате модификации любых данных в базе данных, а не только данных таблицы, с которой работает пользователь.
- На значении переменной отражаются любые изменения временной отметки в базе данных, а не только изменения, происходящие в текущем соединении

Таким образом, не следует рассчитывать на то, что значение рассматриваемой системной переменной действительно является результатом последних по времени действий, выполненных в приложении (в связи с тем, что, возможно, наряду с этим в базе данных осуществлялись какие-то другие действия, которые привели к изменению этого значения), поэтому сам автор не находит для нее достаточно широкую область применения.

Системная переменная @@ERROR

Системная переменная `@@ERROR` содержит код ошибки, относящийся к последнему оператору T-SQL, который был выполнен в текущем соединении. Если ошибка не возникла, эта системная переменная содержит значение `0`.

Системная переменная `@@ERROR` становится особенно важной при создании хранимых процедур или триггеров; в этом случае без ее использования практически невозможно обойтись.

Следует помнить, что полученное значение системной переменной `@@ERROR` относится только к последнему выполненному оператору. Это означает, что в случае необходимости провести проверку на наличие ошибки после выполнения какого-то конкретного оператора такая проверка должна быть предусмотрена в непосредственно следующим за ним операторе; еще одна возможность сохранить значение этой переменной состоит в том, чтобы присвоить его какой-то промежуточной переменной.

Со списком всех системных ошибок можно ознакомиться с помощью системной таблицы `sysmessages` базы данных `master`.

Для создания собственных пользовательских сообщений об ошибках можно воспользоваться процедурой `sp_addmessage`.

Системная переменная @@FETCH_STATUS

Системная переменная `@@FETCH_STATUS` содержит индикатор состояния последней операции `FETCH` с курсором.

При использовании курсоров практически невозможно обойтись без системной переменной `@@FETCH_STATUS`. Именно эта переменная позволяет определить, успешной или неудачной была попытка перейти к определенной строке в курсоре. Значения этой переменной представляют собой константы, присваиваемые переменной с учетом того, была ли в СУБД SQL Server успешно выполнена последняя операция `FETCH` с курсором, а если операция `FETCH` окончилась неудачей, позволяет узнать причину ошибки. Эти константы перечислены ниже.

- `0`. Успешное завершение.
- `-1`. Неудачное завершение, связанное с тем, что указатель курсора находится за пределами набора данных курсора (перед началом или вслед за концом набора данных).
- `-2`. Неудачное завершение. Была предпринята попытка выборки строки, но эта строка не найдена. Обычно такая ситуация возникает в связи с тем, что строка удалена в промежутке времени между созданием набора данных курсора и попыткой перейти к этой строке. Как правило, такая ошибка возникает только в прокручиваемых, нединамических курсорах.

Для удобства чтения обычно рекомендуется объявлять некоторые константы перед использованием системной переменной @@FETCH_STATUS.

Примеры подобных объявлений приведены ниже.

```
DECLARE @NOTFOUND int
DECLARE @BEGINEND int
SELECT @NOTFOUND = -2
SELECT @BEGINEND = -1
```

Это позволяет применять подобные константы в конструкции WHILE оператора обработки курсора в цикле, а не просто задавать целое число. Благодаря этому удобство чтения кода значительно повышается.

Системная переменная @@IDENTITY

Системная переменная @@IDENTITY содержит последнее идентификационное значение, созданное в текущем соединении.

Если в процессе работы осуществляется вставка идентификационных значений в столбцы идентификации, а затем эти значения используются для формирования ссылки на исходную таблицу в другой таблице (в качестве внешнего ключа), то приходится постоянно использовать системную переменную @@IDENTITY. При этом создается родительская строка (обычно содержащая значение типа identity, подлежащее выборке), а затем осуществляется выборка значения @@IDENTITY для определения того, какое значение ключа должно применяться для обеспечения связи между родительской и дочерней строками.

Если осуществляется вставка данных с применением идентификационных значений в несколько таблиц, то следует помнить, что системная переменная @@IDENTITY всегда содержит значение последнего вставленного идентификационного значения; все промежуточные значения теряются, если не предусмотрено их присваивание каким-то промежуточным переменным после выполнения каждой операции вставки. Кроме того, если последняя строка, в которую была выполнена вставка, не содержит столбца идентификации, то системная переменная @@IDENTITY содержит NULL-значение.

Системная переменная @@IDLE

Системная переменная @@IDLE содержит значение времени в миллисекундах (точность которого зависит от степени дискретизации показаний системного таймера), в течение которого СУБД SQL Server простаивала со времени своего последнего запуска.

Системная переменная @@IDLE может рассматриваться в своем роде как противоположная системной переменной @@CPU_BUSY. По существу, она содержит значение, которое позволяет узнать о том, сколько времени было израсходовано СУБД SQL Server в состоянии простоя. Если кто-либо сумеет найти рациональное применение в программе для этой переменной, пусть отправит мне письмо по электронной почте; я был бы рад узнать, что это возможно (сам я не смог придумать ни одного способа ее применения).

Системная переменная @@IO_BUSY

Системная переменная @@IO_BUSY содержит значение времени в миллисекундах (точность которого зависит от степени дискретизации показаний системного таймера), затраченного SQL Server на выполнение операций ввода и вывода со времени последнего запуска. Это значение сбрасывается после каждого перезапуска СУБД SQL Server.

Фактически при использовании этой системной переменной не возникают какие-либо сложности, но она, по мнению автора, также относится к категории переменных, для которых трудно найти в программе рациональное применение.

Системные переменные @@LANGID и @@LANGUAGE

Системные переменные @@LANGID и @@LANGUAGE содержат соответственно идентификатор и имя языка, используемого в настоящее время.

Применение этих переменных представляет собой удобный способ определения того, был ли используемый программный продукт установлен с учетом требований локализации и, в случае положительного ответа на этот вопрос, какой национальный язык применяется по умолчанию.

Для получения полного списка языков, поддерживаемых в настоящее время программным обеспечением SQL Server, воспользуйтесь системной хранимой процедурой sp_helplanguage.

Системная переменная @@LOCK_TIMEOUT

Системная переменная @@LOCK_TIMEOUT содержит текущее значение времени в миллисекундах, в течение которого система ожидает освобождения заблокированного ресурса.

Если какой-либо ресурс (страница, строка, таблица и пр.) заблокирован, процесс, пытающийся получить к нему доступ, останавливается и ожидает освобождения блокировки. Системная переменная @@LOCK_TIMEOUT определяет, как долго процесс должен находиться в состоянии ожидания, прежде чем будет отменена текущая операция.

По умолчанию время ожидания установлено равным 0. Это равносильно требованию устанавливать неопределенно долгую продолжительность ожидания. Иное значение эта переменная может приобрести, только если кто-то изменяет его на уровне системы (с помощью хранимой процедуры sp_configure). Но независимо от значения, заданного в системе по умолчанию, попытка получить значение системной переменной @@LOCK_TIMEOUT приводит к получению -1, если требуемое значение текущего соединения не установлено вручную с помощью оператора SET LOCK_TIMEOUT.

Системная переменная @@MAX_CONNECTIONS

Системная переменная @@MAX_CONNECTIONS содержит данные о максимальном количестве одновременных пользовательских соединений, которые разрешается устанавливать в СУБД SQL Server.

Не следует ошибочно считать, что эти данные означают то же самое, что и данные, содержащиеся в свойстве Maximum Connections, которое можно найти в программе Management Console. Рассматриваемое значение определяется с учетом пра-

вил лицензирования и составляет очень большую величину, если выбран принцип лицензирования "на каждое рабочее место" ("per seat").

Следует также отметить, что фактическое количество разрешенных к применению пользовательских соединений зависит от используемой версии SQL Server, а также от характеристик приложения (приложений) и аппаратных средств.

Системная переменная @@MAX_PRECISION

Системная переменная @@MAX_PRECISION позволяет определить уровень точности, заданный в настоящее время для десятичных и других числовых типов данных.

По умолчанию эта переменная определяет точность, равную 38 позициям, но это значение может быть изменено с помощью опции /p во время запуска СУБД SQL Server. Параметр /p может быть задан при запуске СУБД SQL Server из командной строки или добавлен к параметрам Startup для службы MSSQLServer в аплете Services операционной системы Windows 2000/2003/XP.

Системная переменная @@NESTLEVEL

Системная переменная @@NESTLEVEL содержит данные о текущем уровне вложенности для вложенных хранимых процедур.

Для первой хранимой процедуры, подлежащей запуску, значение @@NESTLEVEL равно 0. Если эта хранимая процедура вызывает другую, то вторая хранимая процедура рассматривается как вложенная в первую хранимую процедуру (а значение @@NESTLEVEL увеличивается и становится равным 1). Аналогичным образом, вторая хранимая процедура может вызывать третью и т.д. (вплоть до максимальной глубины вложенности, равной 32 уровням). Если в приложении достигается уровень глубины вложенности больше 32, происходит аварийное завершение транзакции. Из этого следует, что проект подобного приложения должен быть пересмотрен.

Системная переменная @@OPTIONS

Системная переменная @@OPTIONS предоставляет информацию о том, какие опции заданы с помощью команды SET.

Поскольку системная переменная @@OPTIONS содержит только одно значение, а количество опций, которые могут быть заданы с помощью команды SET, достаточно велико, то в программном обеспечении SQL Server для указания на то, какие значения установлены, используются двоичные флажки. Чтобы проверить, была ли задана интересующая вас опция, необходимо применить к значению опции оператор побитовой обработки, например, следующим образом:

```
IF (@@OPTIONS & 2)
```

Если выражение @@OPTIONS & 2 принимает значение True, можно сделать вывод, что для текущего соединения задана опция IMPLICIT_TRANSACTIONS. Значения двоичных флажков перечислены в табл. Б.1.

Таблица Б.1. Значения двоичных флажков системной переменной @@OPTIONS

Двоичный флажок	Опция SET	Описание
1	DISABLE_DEF_CNST_CHK	Промежуточная или отложенная проверка ограничений
2	IMPLICIT_TRANSACTIONS	Неявный запуск транзакции при выполнении оператора
4	CURSOR_CLOSE_ON_COMMIT	Закрытие курсора после выполнения операции COMMIT
8	ANSI_WARNINGS	Формирование предупреждающих сообщений об усечении таблиц и обнаружении NULL-значений при выполнении агрегирующих функций
16	ANSI_PADDING	Дополнение переменных фиксированной длины
32	ANSI_NULLS	Применяемый способ трактовки NULL-значений во время выполнения операторов сравнения на равенство
64	ARITHABORT	Завершение запроса при обнаружении ошибки переполнения или деления на нуль во время выполнения запроса
128	ARITHIGNORE	Возврат NULL-значений при возникновении ошибки переполнения или деления на нуль во время выполнения запроса
256	QUOTED_IDENTIFIER	Проведение различий между одинарными и двойными кавычками при вычислении любого выражения
512	NOCOUNT	Отмена вывода сообщения row(s) affected в конце возвращаемых данных любого оператора
1024	ANSI_NULL_DFLT_ON	Применение в сеансе средств обеспечения совместимости со стандартом ANSI в части поддержки NULL-значений. Столбцы, создаваемые в составе новых таблиц или добавляемые в существующие таблицы без явного указания параметров поддержки NULL-значений, определяются как допускающие использование NULL-значений. Эта опция является взаимоисключающей с опцией ANSI_NULL_DFLT_OFF
2048	ANSI_NULL_DFLT_OFF	Отказ от применения в сеансе средств обеспечения совместимости со стандартом ANSI в части поддержки NULL-значений. Столбцы, создаваемые в составе новых таблиц или добавляемые в существующие таблицы без явного указания параметров поддержки NULL-значений, определяются как не допускающие использование NULL-значений. Эта опция является взаимоисключающей с опцией ANSI_NULL_DFLT_ON
4096	CONCAT_NULL_YIELDS_NULL	Возврат NULL-значений при выполнении операций конкатенации NULL-значений со строковыми значениями
8192	NUMERIC_ROUNDABORT	Выработка сообщения об ошибке при обнаружении потери точности в выражении

Системные переменные @@PACK_RECEIVED и @@PACK_SENT

Системные переменные @@PACK_RECEIVED и @@PACK_SENT позволяют получить данные о количестве входных и выходных пакетов, переданных и полученных из сети сервером SQL Server, после его последнего запуска.

Как правило, значения этих переменных используются в инструментальных средствах устранения нарушений в работе сети.

Системная переменная @@PACKET_ERRORS

Системная переменная @@PACKET_ERRORS содержит информацию о количестве ошибок сетевых пакетов, которые обнаружены в соединениях с СУБД SQL Server со времени последнего запуска SQL Server.

Как правило, значение этой переменной используется в инструментальных средствах устранения нарушений в работе сети.

Системная переменная @@PROCID

Системная переменная @@PROCID содержит значение идентификатора выполняющейся в настоящее время хранимой процедуры.

Как правило, значение этой переменной используется в инструментальных средствах устранения нарушений в работе процесса, в котором используется большое количество ресурсов. Главным образом предназначена для использования администратором базы данных.

Системная переменная @@REMSERVER

Системная переменная @@REMSERVER содержит информацию о сервере (в том виде, в каком она обнаруживается во время регистрации учетной записи), относящуюся к серверу, с которого вызвана хранимая процедура.

Системная переменная @@REMSERVER используется только в хранимых процедурах. Она является удобным средством такой организации работы, в которой требуется, чтобы хранимая процедура действовала по-разному, в зависимости от того, с какого удаленного сервера она вызвана (при этом часто учитывается географическое местонахождение).

Системная переменная @@ROWCOUNT

Системная переменная @@ROWCOUNT содержит информацию о количестве строк, затронутых последним оператором.

Эта системная переменная относится к числу наиболее широко используемых. В частности, она обычно применяется для проверки на наличие ошибок, отличных от ошибок этапа прогона. Под этим подразумеваются такие ошибки, которые являются следствием использования в программе неправильных алгоритмов, но сама СУБД SQL Server не обнаруживает при этом никаких нарушений в работе. В качестве примера можно указать такую ситуацию, в которой осуществляется обновление данных с учетом некоторого условия, но обнаруживается, что в этом обновлении затрагивается нулевое количество строк. Вполне очевидно, что если в клиентской программе вы-

зван на выполнение оператор модификации определенных строк, то предполагается прежде всего наличие строк, соответствующих заданному критерию. Если же количество затронутых строк равно нулю, это может свидетельствовать о том, что допущена какая-то ошибка.

С другой стороны, системная переменная `@@ROWCOUNT` возвращает также значение `0` при выполнении любого оператора, в котором не предусматривается обработка строк.

Системная переменная @@SERVERNAME

Системная переменная `@@SERVERNAME` содержит имя локального сервера, с которого был запущен на выполнение данный сценарий.

Если на компьютере инсталлировано несколько экземпляров СУБД SQL Server (в качестве примера такой организации работы можно назвать службу Web-хостинга, в которой для каждого клиента используется отдельная инсталляция SQL Server), то системная переменная `@@SERVERNAME` возвращает информацию, касающуюся имени локального сервера, приведенную в табл. Б.2, если имя локального сервера не было изменено со времени инсталляции.

Таблица Б.2. Информация об имени локального сервера

Экземпляр	Информация о сервере
Экземпляр, применяемый по умолчанию	`<servername>`
Именованный экземпляр	`<servername\instancename>`
Виртуальный сервер — экземпляр, применяемый по умолчанию	`<virtualservername>`
Виртуальный сервер — именованный экземпляр	`<virtualservername\instancename>`

Системная переменная @@SERVICENAME

Системная переменная `@@SERVICENAME` содержит имя ключа реестра, под которым эксплуатируется СУБД SQL Server.

Эта системная переменная содержит определенные данные только в операционной системе Windows 2000/2003/XP и (в любой из этих операционных систем) должна всегда иметь значение `MSSQLService`, если это значение не было модифицировано в системном реестре преднамеренно.

Системная переменная @@SPID

Системная переменная `@@SPID` содержит идентификатор серверного процесса (Server Process ID — SPID) текущего пользовательского процесса.

Эта системная переменная содержит такой же идентификатор процесса, который обнаруживается после запуска хранимой процедуры `sp_who`. Один из способов применения этой системной переменной состоит в том, что она позволяет узнать значение SPID текущего соединения, которое может использоваться администратором базы данных для контроля над выполнением текущей задачи и в случае необходимости — для ее завершения.

Системная переменная @@TEXTSIZE

Системная переменная @@TEXTSIZE содержит текущее значение опции TEXTSIZE, установленной с помощью оператора SET, которая определяет максимальную длину в байтах, возвращаемую оператором SELECT при выборке текстовых данных или данных изображения.

По умолчанию значение этой переменной составляет 4096 байтов (4 Кбайт). Это значение можно изменить с использованием оператора SET TEXTSIZE.

Системная переменная @@TIMETICKS

Системная переменная @@TIMETICKS содержит информацию о том, чему равна продолжительность тактового интервала в микросекундах. Соответствующее значение зависит от аппаратного обеспечения компьютера. Вообще говоря, системная переменная @@TIMETICKS также относится к категории программных средств, для которых трудно найти рациональное применение.

Системная переменная @@TOTAL_ERRORS

Системная переменная @@TOTAL_ERRORS содержит информацию о количестве ошибок чтения-записи на диске, обнаруженных в СУБД SQL Server со времени последнего запуска.

Не следует путать ошибки чтения-записи с ошибками этапа прогона или считать, что эта переменная имеет какое-то отношение к переменной @@ERROR. Она касается исключительно тех проблем, которые возникают во время выполнения физических операций ввода-вывода. Системную переменную @@TOTAL_ERRORS также можно считать относящейся к категории программных средств, для которых трудно найти рациональное применение. Главным образом ее можно было бы использовать в системных диагностических сценариях, но для этой цели, по-видимому, больше подходят средства программы Performance Monitor.

Системные переменные @@TOTAL_READ и @@TOTAL_WRITE

Системные переменные @@TOTAL_READ и @@TOTAL_WRITE позволяют соответственно определить общее количество операций чтения–записи на диске, обнаруженных в СУБД SQL Server со времени ее последнего запуска.

Имена этих переменных немного не соответствуют их назначению, поскольку они не содержат информацию об операциях чтения-записи в кэше, а предоставляют только сведения о физических операциях ввода-вывода.

Системная переменная @@TRANCOUNT

Системная переменная @@TRANCOUNT содержит информацию о количестве активных транзакций (по существу, об уровне вложенности транзакций) для текущего соединения.

Эта системная переменная становится очень важной, если в приложении применяется транзакционная обработка. Как правило, использование вложенных транзакций не рекомендуется, но в некоторых обстоятельствах без них трудно обойтись.

В таком случае важно иметь информацию о том, на каком уровне выполняется текущая транзакция (например, если используется алгоритм, предусматривающий запуск транзакции только при том условии, что в настоящее время не выполняется какая-то другая транзакция).

Если в данный момент не выполняется какая-либо транзакция, то значение @@TRANCOUNT равно 0. В связи с этим рассмотрим следующий краткий пример:

```
SELECT @@TRANCOUNT As TransactionNestLevel      -- В данный момент уровень
                        -- вложенности транзакции должен быть равен нулю
BEGIN TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      -- В данный момент уровень
                        -- вложенности транзакции должен быть равен единице
  BEGIN TRAN
    SELECT @@TRANCOUNT As TransactionNestLevel  -- В данный момент уровень
                        -- вложенности транзакции должен быть равен двум
  COMMIT TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      -- В данный момент уровень
             -- вложенности транзакции должен снова возвратиться к единице
ROLLBACK TRAN
SELECT @@TRANCOUNT As TransactionNestLevel      -- В данный момент уровень
            -- вложенности транзакции должен снова возвратиться к нулю
```

Обратите внимание на то, что в данном примере значение @@TRANCOUNT также достигло бы нуля в самом конце, если бы в последней транзакции был выполнен оператор COMMIT.

Системная переменная @@VERSION

Системная переменная @@VERSION позволяет получить информацию о текущей версии SQL Server, а также о типе процессора и архитектуре операционной системы.

Например, выполнение оператора

```
SELECT @@VERSION
```

приводит к получению такого результата:

```
------------------------------------------------------------------------------
Microsoft SQL Server 2005 - 9.00.1116 (Intel X86)
  Apr  9 2005 20:56:37
  Copyright (c) 1988-2004 Microsoft Corporation
  Beta Edition on Windows NT 5.1 (Build 2600: Service Pack 2)

(1 row(s) affected)
```

К сожалению, эти данные не представлены в виде какого-либо структурированного значения, разбитого на отдельные поля, поэтому для получения конкретной информации необходимо выполнить синтаксический анализ значения этой переменной.

Вместо указанной системной переменной рекомендуется использовать системную хранимую процедуру xp_msver; эта процедура возвращает информацию в таком виде, что задача выборки из полученных результатов какой-то конкретной информации становится проще.

Агрегирующие функции

Агрегирующие функции применяются к наборам строк, а не к отдельным строкам. Информация, содержащаяся в многочисленных строках, обрабатывается каким-то определенным способом, а затем оформляется в виде результирующих данных, состоящих из одной строки. Агрегирующие функции часто используются вместе с конструкцией GROUP BY.

Список агрегирующих функций приведен ниже.

- AVG.
- CHECKSUM.
- CHECKSUM_AGG.
- COUNT.
- COUNT_BIG.
- GROUPING.
- MAX.
- MIN.
- STDEV.
- STDEVP.
- SUM.
- VAR.
- VARP.

При вызове большинства агрегирующих функций может использоваться ключевое слово ALL или DISTINCT. Параметр ALL применяется по умолчанию и указывает на то, что действие функции должно распространяться на все значения в выражении, даже если одно и то же значение появляется несколько раз. С другой стороны, параметр DISTINCT означает, что значение должно быть включено в функцию только один раз, даже если обнаруживается несколько дубликатов этого значения.

Вложение вызовов агрегирующих функций не допускается. Параметр <expression> не может представлять собой подзапрос.

Функция AVG

Функция AVG возвращает арифметическое среднее значений, представленных в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
AVG([ALL | DISTINCT] <expression>)
```

Параметр <expression> должен содержать числовые значения. Неопределенные значения игнорируются.

Функция COUNT

Функция COUNT возвращает данные о количестве элементов, представленных в параметре <expression>. Возвращаемые данные относятся к типу int. Для вызова этой функции применяется следующий синтаксис:

```
COUNT
(
   [ALL | DISTINCT] <expression> | *
)
```

Параметр <expression> не может относиться к типу данных uniqueidentifier, text, image или ntext. При использовании значения параметра * происходит возврат данных о количестве строк в таблице; при этом дублирующиеся значения или NULL-значения не исключаются.

Функция COUNT_BIG

Функция COUNT_BIG возвращает данные о количестве элементов в группе. Эта функция весьма напоминает функцию COUNT, описанную выше, если не считать того, что возвращаемое значение имеет тип данных bigint. Для вызова этой функции применяется следующий синтаксис:

```
COUNT_BIG
(
   [ALL | DISTINCT ] <expression> | *
)
```

Функция GROUPING

Функция GROUPING добавляет дополнительный столбец к выводу оператора SELECT. Функция GROUPING используется в сочетании с конструкциями CUBE или ROLLUP, чтобы можно было провести различие между обычными NULL-значениями и NULL-значениями, полученными в результате выполнения операций CUBE и ROLLUP. Для вызова этой функции применяется следующий синтаксис:

```
GROUPING (<column_name>)
```

Функция GROUPING используется только в списке выборки. Ее параметром является столбец с именем <column_name>, который используется в конструкции GROUP BY и в котором должно быть проверено наличие NULL-значений.

Функция MAX

Функция MAX возвращает максимальное среди значений, представленных в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
MAX([ALL | DISTINCT] <expression>)
```

При вычислении функции MAX все NULL-значения игнорируются.

Функция MIN

Функция MAX возвращает минимальное среди значений, представленных в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
MIN([ALL | DISTINCT] <expression>)
```

При вычислении функции MIN все NULL-значения игнорируются.

Функция STDEV

Функция STDEV возвращает результат вычисления среднеквадратичного отклонение по всем значениям, представленным в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
STDEV(<expression>)
```

При вычислении функции STDEV все NULL-значения игнорируются.

Функция STDEVP

Функция STDEVP возвращает среднеквадратичное отклонение для выборки, состоящей из всех значений, представленных в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
STDEVP(<expression>)
```

При вычислении функции STDEVP все NULL-значения игнорируются.

Функция SUM

Функция SUM возвращает сумму всех значений, представленных в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
SUM([ALL | DISTINCT] <expression>)
```

При вычислении функции SUM все NULL-значения игнорируются.

Функция VAR

Функция VAR возвращает результат вычисления дисперсии по всем значениям, представленным в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
VAR(<expression>)
```

При вычислении функции VAR все NULL-значения игнорируются.

Функция VARP

Функция VARP возвращает дисперсию для выборки, состоящей из всех значений, представленных в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
VARP(<expression>)
```

При вычислении функции VARP все NULL-значения игнорируются.

Функции для работы с курсорами

Для работы с курсорами предусмотрена только одна функция (CURSOR_STATUS), которая предоставляет информацию о курсорах.

Функция CURSOR_STATUS

Функция CURSOR_STATUS позволяет определить в программе, из которой вызвана хранимая процедура, был ли возвращен этой процедурой курсор и результирующий набор. Для вызова этой функции применяется следующий синтаксис:

```
CURSOR_STATUS
   (
     {'<local>', '<cursor_name>'}
     | {'<global'>, '<cursor_name>'}
     | {'<variable>', '<cursor_variable>'}
   )
```

Параметры <local>, <global> и <variable> определяют константы, которые указывают, откуда поступают данные для курсора. Параметр <local> обозначает имя локального курсора, параметр <global> — имя глобального курсора, а параметр <variable> — локальную переменную.

Если используется форма вызова функции с параметром <cursor_name>, то возможно получение одного из четырех описанных ниже возвращаемых значений.

- 1. Курсор открыт. Если курсор является динамическим, его результирующий набор содержит от нуля или больше строк. Если курсор не является динамическим, то результирующий набор содержит от одной или больше строк.
- 0. Результирующий набор курсора пуст.
- -1. Курсор закрыт.
- -3. Курсор с именем '<cursor_name>' не существует.

Если используется форма вызова функции с параметром <cursor_variable>, то возможно получение одного из пяти описанных ниже возвращаемых значений.

- 1. Курсор открыт. Если курсор является динамическим, его результирующий набор содержит от нуля или больше строк. Если курсор не является динамическим, то результирующий набор содержит от одной или больше строк.
- 0. Результирующий набор курсора пуст.
- -1. Курсор закрыт.
- -2. Курсор, присвоенный переменной <cursor_variable>, не существует.
- -3. Переменная с именем <cursor_variable> не существует, и если даже она существует, то ей еще не присвоен курсор.

Функции для работы со значениями даты и времени

Функции для работы со значениями даты и времени выполняют операции со значениями, которые относятся к типам данных datetime и smalldatetime или представляют собой символьные данные в форме даты. Перечень этих функций приведен ниже.

- DATEADD.
- DATEDIFF.

- DATENAME.
- DATEPART.
- DAY.
- GETDATE.
- GETUTCDATE.
- MONTH.
- YEAR.

Программное обеспечение SQL Server распознает одиннадцать компонентов обозначения даты и соответствующих аббревиатур (табл. Б.3).

Таблица Б.3. Компоненты обозначения даты и соответствующие аббревиатуры

Компонент обозначения даты	Аббревиатура	Описание
year	yy, yyyy	Год
quarter	qq, q	Квартал
month	mm, m	Месяц
dayofyear	dy, y	Номер дня в году
day	dd, d	День месяца
week	wk, ww	Неделя
weekday	dw	День недели
hour	hh	Час
minute	mi, n	Минута
second	ss, s	Секунда
millisecond	ms	Миллисекунда

Функция DATEADD

Функция DATEADD складывает со значением даты <date> указанный интервал <number> и возвращает новое значение даты. Для вызова этой функции применяется следующий синтаксис:

```
DATEADD(<datepart>, <number>, <date>)
```

Параметр <datepart> определяет единицу измерения интервала (day, week, month и т.д.) и может принимать значение любого из компонентов обозначения даты, распознаваемых СУБД SQL Server. Параметр <number> задает количество компонентов обозначения даты, которые должны быть добавлены к дате <date>.

Функция DATEDIFF

Функция DATEDIFF возвращает результат вычисления разности между двумя указанными датами в указанных единицах измерения времени (таких как часы, сутки и недели). Для вызова этой функции применяется следующий синтаксис:

```
DATEDIFF(<datepart>, <startdate>, <enddate>)
```

Параметр <datepart> может принимать значение любого из компонентов обозначения даты, распознаваемых СУБД SQL Server, и задает используемую единицу измерения времени.

Функция DATENAME

Функция DATENAME возвращает строку, представляющую имя указанного компонента обозначения даты <datepart> (в качестве компонента обозначения даты может быть, например, указано "1999", "Thursday", "July") указанной даты <date>. Для вызова этой функции применяется следующий синтаксис:

```
DATENAME(<datepart>, <date>)
```

Функция DATEPART

Функция DATEPART возвращает целое число, которое представляет указанный компонент обозначения даты <datepart> указанной даты <date>. Для вызова этой функции применяется следующий синтаксис:

```
DATEPART(<datepart>, <date>)
```

Функция DAY эквивалентна DATEPART(dd, <date>), MONTH эквивалентна DATEPART(mm, <date>), а YEAR эквивалентна DATEPART(yy, <date>).

Функция DAY

Функция DAY возвращает целое число, представляющее часть указанной даты <date>, соответствующую дню месяца. Для вызова этой функции применяется следующий синтаксис:

```
DAY(<date>)
```

Функция DAY эквивалентна DATEPART(dd, <date>).

Функция GETDATE

Функция GETDATE возвращает текущую системную дату и времени. Для вызова этой функции применяется следующий синтаксис:

```
GETDATE()
```

Функция GETUTCDATE

Функция GETUTCDATE возвращает текущее значение всеобщего скоординированного времени (Universal Coordinated Time — UTC). Другими словами, это функция возвращает значение всемирного гринвичского среднего времени. Для получения этого значения используется значение местного времени, отсчитываемое сервером, которое складывается со значением часового пояса, на основании чего вычисляется всеобщее скоординированное время; при этом учитывается сезонное время. Функцию GETUTCDATE нельзя вызывать из пользовательской функции. Для вызова этой функции применяется следующий синтаксис:

```
GETUTCDATE()
```

Функция MONTH

Функция MONTH возвращает целое число, представляющее часть указанной даты <date>, соответствующую номеру месяца года. Для вызова этой функции применяется следующий синтаксис:

```
MONTH(<date>)
```

Функция MONTH эквивалентна DATEPART(mm, <date>).

Функция YEAR

Функция YEAR возвращает целое число, представляющее часть указанной даты <date>, соответствующую числовому значению года. Для вызова этой функции применяется следующий синтаксис:

```
YEAR(<date>)
```

Функция YEAR эквивалентна DATEPART(yy, <date>).

Математические функции

Математические функции применяются для выполнения вычислений. Перечень этих функций приведен ниже.

- ABS.
- ACOS.
- ASIN.
- ATAN.
- ATN2.
- CEILING.
- COS.
- COT.
- DEGREES.
- EXP.
- FLOOR.
- LOG.
- LOG10.
- PI.
- POWER.
- RADIANS.
- RAND.
- ROUND.
- SIGN.
- SIN.
- SQRT.
- SQUARE.
- TAN.

Функция ABS

Функция ABS возвращает положительное, абсолютное значение числового параметра <numeric expression>, применяемого в качестве параметра. Для вызова этой функции применяется следующий синтаксис:

```
ABS(<numeric expression>)
```

Функция ACOS

Функция ACOS возвращает значение угла в радианах, косинус которого задан с помощью параметра <expression> (другими словами, эта функция возвращает арккосинус параметра <expression>). Для вызова этой функции применяется следующий синтаксис:

```
ACOS(<expression>)
```

Значение параметра <expression> должно находиться в пределах между -1 и 1 и относиться к типу данных float.

Функция ASIN

Функция ASIN возвращает значение угла в радианах, синус которого задан с помощью параметра <expression> (другими словами, эта функция возвращает арксинус параметра <expression>). Для вызова этой функции применяется следующий синтаксис:

```
ASIN(<expression>)
```

Значение параметра <expression> должно находиться в пределах между -1 и 1 и относиться к типу данных float.

Функция ATAN

Функция ATAN возвращает значение угла в радианах, тангенс которого задан с помощью параметра <expression> (другими словами, это функция возвращает арктангенс параметра <expression>). Для вызова этой функции применяется следующий синтаксис:

```
ATAN(<expression>)
```

Параметр <expression> должен иметь тип данных float.

Функция ATN2

Функция ATN2 возвращает значение угла в радианах, тангенс которого задан параметрами <expression1> и <expression2> (другими словами, эта функция арктангенса этих двух параметров). Для вызова этой функции применяется следующий синтаксис:

```
ATN2(<expression1>, <expression2>)
```

Оба параметра, и <expression1>, и <expression2>, должны иметь тип данных float.

Функция CEILING

Функция CEILING возвращает значение наименьшего целого числа, которое больше или равно указанному параметру <expression>. Для вызова этой функции применяется следующий синтаксис:

```
CEILING(<expression>)
```

Функция COS

Функция COS возвращает косинус угла, заданного параметром <expression>. Для вызова этой функции применяется следующий синтаксис:

```
COS(<expression>)
```

Угол должен быть указан в радианах, а параметр <expression> должен иметь тип данных float.

Функция COT

Функция COT возвращает котангенс угла, заданного параметром <expression>. Для вызова этой функции применяется следующий синтаксис:

```
COT(<expression>)
```

Угол должен быть указан в радианах, а параметр <expression> должен иметь тип данных float.

Функция DEGREES

Функция DEGREES принимает в качестве параметра угол, заданный в радианах (параметр <expression>), и возвращает значение угла в градусах. Для вызова этой функции применяется следующий синтаксис:

```
DEGREES(<expression>)
```

Функция EXP

Функция EXP возвращает экспоненциальное значение данных, представленных в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
EXP(<expression>)
```

Параметр <expression> должен иметь тип данных float.

Функция FLOOR

Функция FLOOR возвращает значение наибольшего целого числа, которое меньше или равно указанному параметру <expression>. Для вызова этой функции применяется следующий синтаксис:

```
FLOOR(<expression>)
```

Функция LOG

Функция LOG возвращает натуральный логарифм значения, заданного параметром <expression>. Для вызова этой функции применяется следующий синтаксис:

```
LOG(<expression>)
```

Параметр <expression> должен иметь тип данных float.

Функция LOG10

Функция LOG10 возвращает логарифм по основанию 10 значения, заданного параметром <expression>. Для вызова этой функции применяется следующий синтаксис:

```
LOG10(<expression>)
```

Параметр <expression> должен иметь тип данных float.

Функция PI

Функция PI возвращает значение константы π. Для вызова этой функции применяется следующий синтаксис:

```
PI()
```

Функция POWER

Функция POWER возводит значение указанного параметра <expression> в указанную степень. Для вызова этой функции применяется следующий синтаксис:

```
POWER(<expression>, <power>)
```

Функция RADIANS

Функция RADIANS возвращает значение угла в радианах, соответствующее значению угла в градусах, заданному в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
RADIANS(<expression>)
```

Функция RAND

Функция RAND возвращает случайное значение, находящееся в пределах между 0 и 1. Для вызова этой функции применяется следующий синтаксис:

```
RAND([<seed>])
```

Значение параметра <seed> представляет собой целочисленное выражение, которое определяет начальное значение.

Функция ROUND

Функция ROUND принимает число, заданное в параметре <expression>, и округляет его с учетом указанной точности:

```
ROUND(<expression>, <length> [, <function>])
```

Параметр <length> определяет точность, до которой должно быть округлено число, заданное параметром <expression>. Параметр <length> должен иметь тип данных tinyint, smallint или int. Для определения того, должно ли быть число округлено или усечено, может использоваться необязательный параметр <function>. Если значение параметра <function> не задано или равно 0 (это значение применяется по умолчанию), то значение, представленное в параметре <expression>, округляется. Если же в качестве параметра <function> задано какое-либо другое значение, кроме 0, то значение, представленное в параметре <expression>, будет усечено.

Функция SIGN

Функция SIGN возвращает знак числа, представленного параметром <expression>. Возможными возвращаемыми значениями являются +1 (обозначает положительное число), 0 (обозначает нуль) и -1 (обозначает отрицательное число). Для вызова этой функции применяется следующий синтаксис:

```
SIGN(<expression>)
```

Функция SIN

Функция SIN возвращает синус заданного угла. Для вызова этой функции применяется следующий синтаксис:

```
SIN(<angle>)
```

Параметр <angle> с обозначением угла должен быть задан в радианах и иметь тип данных float. Возвращаемое значение также будет иметь тип данных float.

Функция SQRT

Функция SQRT возвращает результат вычисления квадратного корня из значения, заданного в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
SQRT(<expression>)
```

Параметр <expression> должен иметь тип данных float.

Функция SQUARE

Функция SQUARE возвращает квадрат значения, заданного в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
SQUARE(<expression>)
```

Параметр <expression> должен иметь тип данных float.

Функция TAN

Функция TAN возвращает тангенс значения, заданного в параметре <expression>. Для вызова этой функции применяется следующий синтаксис:

```
TAN(<expression>)
```

Параметр <expression> задает число в радианах и должен иметь тип данных real или float.

Функции для работы с метаданными

Функции для работы с метаданными позволяют получить информацию о базе данных и ее объектах. Перечень этих функций приведен ниже.

- COL_LENGTH.
- COL_NAME.
- COLUMNPROPERTY.
- DATABASEPROPERTY.
- DATABASEPROPERTYEX.
- DB_ID.
- DB_NAME.
- FILE_ID.
- FILE_NAME.
- FILEGROUP_ID.
- FILEGROUP_NAME.
- FILEGROUPPROPERTY.
- FILEPROPERTY.
- fn_listextendedproperty.
- FULLTEXTCATALOGPROPERTY.
- FULLTEXTSERVICEPROPERTY.
- INDEX_COL.
- INDEXKEY_PROPERTY.
- INDEXPROPERTY.
- OBJECT_ID.
- OBJECT_NAME.
- OBJECTPROPERTY.
- OBJECTPROPERTYEX.
- SQL_VARIANT_PROPERTY.
- TYPE_ID.
- TYPE_NAME.
- TYPEPROPERTY.

Функция COL_LENGTH

Функция COL_LENGTH возвращает данные о длине столбца, соответствующие его определению. Для вызова этой функции применяется следующий синтаксис:

```
COL_LENGTH('<table>', '<column>')
```

Параметр '<column>' позволяет указать имя столбца, для которого должна быть определена длина, а параметр '<table>' задает имя таблицы, которая содержит этот столбец.

Функция COL_NAME

Функция COL_NAME принимает в качестве параметров идентификационный номер таблицы и идентификационный номер столбца и возвращает имя столбца таблицы базы данных. Для вызова этой функции применяется следующий синтаксис:

```
COL_NAME(<table_id>, <cclumn_id>)
```

Параметр <column_id> определяет идентификационный номер столбца, а параметр <table_id> определяет идентификационный номер таблицы, которая содержит этот столбец.

Функция COLUMNPROPERTY

Функция COLUMNPROPERTY возвращает данные о столбце или параметре процедуры. Для вызова этой функции применяется следующий синтаксис:

```
COLUMNPROPERTY(<id>, <column>, <property>)
```

Параметр <id> определяет идентификатор таблицы или процедуры; параметр <column> определяет имя столбца или параметра; параметр <property> определяет данные о столбце или параметре процедуры, которые должны быть возвращены. Параметр <property> может иметь одно из значений, приведенных ниже.

- AllowsNull. Допускает использование NULL-значений.
- IsComputed. Рассматриваемый столбец является вычисляемым.
- IsCursorType. Процедура относится к типу CURSOR.
- IsFullTextIndexed. На столбце задан полнотекстовый индекс.
- IsIdentity. Столбец представляет собой столбец IDENTITY.
- IsIdNotForRepl. По столбцу осуществляется проверка условия IDENTITY NOT FOR REPLICATION.
- IsOutParam. Рассматриваемый параметр процедуры является выходным параметром.
- IsRowGuidCol. Рассматриваемый столбец представляет собой столбец ROWGUIDCOL.
- Precision. Точность, применимая для типа данных столбца или параметра.
- Scale. Масштаб, применимый для типа данных столбца или параметра.
- UseAnsiTrim. Во время создания таблицы параметр настройки заполнения ANSI имел значение ON.

Функция COLUMNPROPERTY возвращает значение 1 (которое соответствует значению True), 0 (False) или NULL (если задан недопустимый параметр). Исключение составляют параметры Precision (при использовании которого функция возвращает данные о точности, применимой для типа данных столбца или параметра) и Scale (при использовании которого функция возвращает данные о масштабе).

Функция DATABASEPROPERTY

Функция DATABASEPROPERTY возвращает параметр конфигурации, соответствующий указанной базе данных и имени свойства. Для вызова этой функции применяется следующий синтаксис:

```
DATABASEPROPERTY('<database>', '<property>')
```

Параметр '<database>' определяет имя базы данных, для которой требуется получить данные об указанном свойстве, а параметр '<property>' содержит имя свойства базы данных и может иметь одно из следующих значений:

- IsAnsiNullDefault. В базе данных поддержка NULL-значений осуществляется в соответствии со стандартом ANSI-92.
- IsAnsiNullsEnabled. Не допускается вычисление значений любых операций сравнения, выполняемых с использованием NULL-значений.
- IsAnsiWarningsEnabled. При обнаружении стандартных условий возникновения ошибок активизируются предупреждающие сообщения.
- IsAutoClose. После выхода из системы последнего пользователя база данных освобождает ресурсы.
- IsAutoShrink. Сжатие файлов базы данных может осуществляться автоматически и периодически.
- IsAutoUpdateStatistics. Разрешено применение опции автоматического обновления статистических данных.
- IsBulkCopy. В базе данных разрешено применение операций, не регистрируемых в журнале (подобных тем, которые осуществляются с помощью программы массового копирования).
- IsCloseCursorsOnCommitEnabled. Любые курсоры, которые остаются открытыми ко времени фиксации транзакции, должны быть закрыты.
- IsDboOnly. Доступ к базе данных разрешен только для владельца базы данных (dbo).
- IsDetached. База данных была отсоединена от сервера в результате выполнения операции отсоединения.
- IsEmergencyMode. База данных находится в аварийном режиме.
- IsFulltextEnabled. В базе данных разрешено применение средств полнотекстового поиска.
- IsInLoad. Выполняется загрузка базы данных.
- IsInRecovery. Происходит восстановление базы данных.
- IsInStandby. База данных находится в режиме только чтения, и разрешено применение журнала восстановления.

- IsLocalCursorsDefault. По умолчанию в объявлениях курсоров применяется опция LOCAL.
- IsNotRecovered. Восстановление базы данных окончилось неудачей.
- IsNullConcat. Применение операций конкатенации с операндами в виде NULL-значений приводит к получению NULL-значений.
- IsOffline. База данных находится в автономном режиме.
- IsQuotedIdentifiersEnabled. Допускается использование двойных кавычек для разграничения идентификаторов.
- IsReadOnly. База данных находится в режиме только чтения.
- IsRecursiveTriggersEnabled. Разрешен рекурсивный запуск триггеров.
- IsShutDown. Во время запуска базы данных возникла проблема.
- IsSingleUser. База данных находится в однопользовательском режиме.
- IsSuspect. База данных находится в подозрительном состоянии.
- IsTruncLog. В базе данных при выполнении контрольных точек усекается журнал.
- Version. Внутренний номер версии кода SQL Server, с применением которого была создана база данных.

Функция DATABASEPROPERTY возвращает значение 1 (которое соответствует значению True), 0 (False) или NULL (если задан недопустимый параметр). Исключение составляют параметр Version (при использовании которого функция возвращает номер версии, если база данных открыта, и NULL, если база данных закрыта).

Функция DATABASEPROPERTYEX

Функция DATABASEPROPERTYEX по существу представляет собой расширение функции DATABASEPROPERTY и возвращает параметр конфигурации, соответствующий указанной базе данных и имени свойства. Для вызова функции DATABASEPROPERTYEX применяется в основном такой же синтаксис, как и для вызова функции DATABASEPROPERTY, который выглядит следующим образом:

```
DATABASEPROPERTYEX('<database>', '<property>')
```

Функция DATABASEPROPERTYEX отличается лишь тем, что поддерживает еще несколько дополнительных свойств, включая описанные ниже.

- Collation. Возвращает данные о схеме упорядочения, применяемой по умолчанию для базы данных. (Напомним, что параметры настройки, которые определяют схему упорядочения, можно также перекрывать на уровне столбца.)
- ComparisonStyle. Указывает стиль сравнения, применяемый в операционной системе Windows в сочетании с конкретной схемой упорядочения (например, чувствительность к регистру).
- IsAnsiPaddingEnabled. Определяет, дополняются ли строки для достижения определенной длины перед выполнением операций сравнения или вставки.
- IsArithmaticAbortEnabled. Определяет, происходит ли аварийное завершение выполнения запросов при обнаружении серьезных ошибок в арифметических операциях (таких как переполнение при обработке данных).

Параметр '<database>' определяет имя базы данных, для которой требуется получить данные об указанном свойстве, а параметр '<property>' содержит имя свойства базы данных и может иметь одно из указанных значений.

Функция DB_ID

Функция DB_ID возвращает значение идентификационного номера базы данных. Для вызова этой функции применяется следующий синтаксис:

```
DB_ID(['<database_name>'])
```

Необязательный параметр '<database_name>' определяет, для какой базы данных требуется получить идентификационный номер. Если параметр '<database_name>' не задан, то вместо указанной базы данных используется текущая база данных.

Функция DB_NAME

Функция DB_NAME возвращает имя базы данных, которая имеет указанный идентификационный номер. Для вызова этой функции применяется следующий синтаксис:

```
DB_NAME([<database_id>])
```

Необязательный параметр <database_id> задает идентификационный номер базы данных, имя которой должно быть возвращено функцией. Если параметр <database_id> не задан, функция возвращает имя текущей базы данных.

Функция FILE_ID

Функция FILE_ID возвращает идентификационный номер файла, относящегося к файлу текущей базы данных с указанным именем. Для вызова этой функции применяется следующий синтаксис:

```
FILE_ID('<file_name>')
```

Параметр '<file_name>' задает имя файла, для которого требуется определить идентификатор.

Функция FILE_NAME

Функция FILE_NAME возвращает имя файла, которое относится к файлу с указанным идентификационным номером файла. Для вызова этой функции применяется следующий синтаксис:

```
FILE_NAME(<file_id>)
```

Параметр <file_id> задает идентификационный номер файла, имя которого требуется определить.

Функция FILEGROUP_ID

Функция FILEGROUP_ID возвращает идентификационный номер файловой группы, относящийся к файловой группе с указанным именем. Для вызова этой функции применяется следующий синтаксис:

```
FILEGROUP_ID('<filegroup_name>')
```

Параметр '<filegroup_name>' задает имя файловой группы, для которой требуется определить идентификатор файловой группы.

Функция FILEGROUP_NAME

Функция FILEGROUP_NAME возвращает имя файловой группы, соответствующее файловой группе с указанным идентификационным номером. Для вызова этой функции применяется следующий синтаксис:

```
FILEGROUP_NAME(<filegroup_id>)
```

Параметр <filegroup_id> задает идентификатор файловой группы, для которой требуется определить имя файловой группы.

Функция FILEGROUPPROPERTY

Функция FILEGROUPPROPERTY после получения таких параметров, как файловая группа и имя свойства, возвращает значение указанного свойства файловой группы. Для вызова этой функции применяется следующий синтаксис:

```
FILEGROUPPROPERTY(<filegroup_name>, <property>)
```

Параметр <filegroup_name> определяет имя файловой группы, которая содержит запрашиваемое свойство, а параметр <property> определяет запрашиваемое свойство и может иметь одно из перечисленных ниже значений.

- IsReadOnly. Файловая группа с указанным именем предназначена только для чтения.
- IsUserDefinedFG. Файловая группа с указанным именем представляет собой определяемую пользователем файловую группу.
- IsDefault. Файловая группа с указанным именем представляет собой файловую группу, применяемую по умолчанию.

Функция FILEGROUPPROPERTY возвращает значение 1 (которое соответствует значению True), 0 (False) или NULL (если задан недопустимый параметр).

Функция FILEPROPERTY

Функция FILEPROPERTY после получения таких параметров, как имя файла и имя свойства, возвращает значение указанного свойства файла группы. Для вызова этой функции применяется следующий синтаксис:

```
FILEPROPERTY(<file_name>, <property>)
```

Параметр <file_name> определяет имя файла, который имеет запрашиваемое свойство, а параметр <property> определяет запрашиваемое свойство и может иметь одно из перечисленных ниже значений.

- IsReadOnly. Файл предназначен только для чтения.
- IsPrimaryFile. Файл является первичным файлом базы данных.
- IsLogFile. Файл представляет собой файл журнала.
- SpaceUsed. Объем пространства, используемого указанным файлом.

Функция FILEGROUPPROPERTY возвращает значение 1 (которое соответствует значению True), 0 (False) или NULL (если задан недопустимый параметр). Исключение составляет параметр SpaceUsed (при использовании которого функция возвращает данные о количестве страниц, распределенных в файле).

Функция FULLTEXTCATALOGPROPERTY

Функция FULLTEXTCATALOGPROPERTY возвращает данные о свойствах полнотекстового каталога. Для вызова этой функции применяется следующий синтаксис:

```
FULLTEXTCATALOGPROPERTY(<catalog_name>, <property>)
```

Параметр <catalog_name> определяет имя полнотекстового каталога, а параметр <property> определяет запрашиваемое свойство. Свойства, для определения которых может быть сделан запрос, перечислены ниже.

- PopulateStatus. Состояние заполнения каталога; возможными возвращаемыми значениями являются 0 (процесс заполнения каталога простаивает), 1 (заполнение продолжается), 2 (заполнение приостановлено), 3 (в ходе заполнения возникли препятствия), 4 (процесс заполнения восстанавливается), 5 (процесс заполнения остановлен), 6 (продолжается инкрементное заполнение) и 7 (происходит обновление индекса).
- ItemCount. Количество полнотекстовых индексированных элементов, находящихся в настоящее время в полнотекстовом каталоге.
- IndexSize. Объем полнотекстового индекса в мегабайтах.
- UniqueKeyCount. Количество уникальных слов, из которых состоит полнотекстовый индекс в этом каталоге.
- LogSize. Объем (в байтах) комбинированного набора журналов регистрации ошибок, связанного с полнотекстовым каталогом.
- PopulateCompletionAge. Разница (в секундах) между временем завершения последней операции заполнения полнотекстового индекса и значением времени 01/01/1990 00:00:00.

Функция FULLTEXTSERVICEPROPERTY

Функция FULLTEXTSERVICEPROPERTY возвращает данные о свойствах, которые определяются на уровне службы полнотекстового поиска. Для вызова этой функции применяется следующий синтаксис:

```
FULLTEXTSERVICEPROPERTY(<property>)
```

Параметр <property> определяет имя запрашиваемого свойства, которое определяется на уровне службы полнотекстового поиска. Параметр <property> может иметь одно из значений, перечисленных ниже.

- ResourceUsage. Возможные значения могут находиться в пределах от 1 (эксплуатируется в фоновом режиме) до 5 (эксплуатируется в монопольном режиме).
- ConnectTimeOut. Время в секундах, в течение которого служба полнотекстового поиска ожидает возобновления активности во всех соединениях с СУБД SQL Server, применяемых для заполнения полнотекстового индекса, прежде чем завершить работу по тайм-ауту.

- IsFulltextInstalled. Принимает значение 1, если на компьютере установлена служба полнотекстового поиска; в противном случае принимает значение 0.

Функция INDEX_COL

Функция INDEX_COL возвращает имя индексированного столбца. Для вызова этой функции применяется следующий синтаксис:

```
INDEX_COL('<table>', <index_id>, <key_id>)
```

Параметр '<table>' определяет имя таблицы, параметр <index_id> задает идентификатор индекса, а параметр <key_id> указывает идентификатор ключа.

Функция INDEXKEY_PROPERTY

Функция INDEXKEY_PROPERTY возвращает информацию о ключе индекса. Для вызова этой функции применяется следующий синтаксис:

```
INDEXKEY_PROPERTY(<table_id>, <index_id>, <key_id>, <property>)
```

Параметр <table_id> представляет собой числовой идентификатор с типом данных int, который определяет проверяемую таблицу. Для получения числового значения <table_id> используется функция OBJECT_ID. Параметр <index_id> определяет идентификатор индекса и имеет типа данных int. Параметр <key_id> определяет позицию столбца с указанным индексом в ключе; например, если при использовании ключа, состоящего из трех столбцов, задано значение параметра <key_id>, равное 2, это означает, что требуется проверить свойства среднего столбца. Наконец, параметр <property> — это заданный в виде строки символов идентификатор одного из двух свойств, значение которого необходимо определить. Двумя возможными значениями параметра <property> являются ColumnId, который возвращает идентификатор столбца, или IsDescending, который возвращает данные о том, в каком порядке осуществляется сортировка значений столбца (1 обозначает убывание, а 0 — возрастание).

Функция INDEXPROPERTY

Функция INDEXPROPERTY после получения таких параметров, как идентификатор таблицы, имя индекса и имя свойства, возвращает значение указанного свойства индекса. Для вызова этой функции применяется следующий синтаксис:

```
INDEXPROPERTY(<table_ID>, <index>, <property>)
```

Параметр <property> определяет свойство индекса, значение которого должно быть определено. Параметр <property> может принимать одно из значений, перечисленных ниже.

- IndexDepth. Глубина индекса.
- IsAutoStatistic. Индекс был создан с применением опции автоматического создания статистических данных хранимой процедуры sp_dboption.
- IsClustered. Индекс является кластеризованным.
- IsStatistics. Индекс был создан с помощью оператора CREATE STATISTICS или опции автоматического создания статистических данных хранимой процедуры sp_dboption.

- IsUnique. Индекс является уникальным.
- IndexFillFactor. В определении индекса задано собственное значение степени заполнения.
- IsPadIndex. В индексе определено пространство, которое должно остаться свободным в каждом внутреннем узле.
- IsFulltextKey. Индекс представляет собой полнотекстовый ключ для таблицы.
- IsHypothetical. Индекс является гипотетическим и не может использоваться непосредственно как путь доступа к данным.

Функция INDEXPROPERTY возвращает значение 1 (которое соответствует значению True), 0 (False) или NULL (если задан недопустимый параметр). Исключение составляют параметры IndexDepth (при использовании которого функция возвращает данные о количестве уровней индекса) и IndexFillFactor (при использовании которого функция возвращает данные о степени заполнения, которая использовалась во время последней операции создания или перестройки индекса).

Функция OBJECT_ID

Функция OBJECT_ID возвращает идентификационный номер указанного объекта базы данных. Для вызова этой функции применяется следующий синтаксис:

```
OBJECT_ID('<object>')
```

Функция OBJECT_NAME

Функция OBJECT_NAME возвращает имя указанного объекта базы данных. Для вызова этой функции применяется следующий синтаксис:

```
OBJECT_NAME(<object_id>)
```

Функция OBJECTPROPERTY

Функция OBJECTPROPERTY возвращает данные об объектах в текущей базе данных. Для вызова этой функции применяется следующий синтаксис:

```
OBJECTPROPERTY(<id>, <property>)
```

Параметр <id> определяет идентификатор требуемого объекта. Параметр <property> указывает, какая информация об объекте должна быть получена. Ниже перечислены допустимые значения свойств.

- CnstIsClustKey.
- CnstIsColumn.
- CnstIsDisabled.
- CnstIsNonclustKey.
- CnstIsNotRepl.
- ExecIsAnsiNullsOn.
- ExecIsDeleteTrigger.

- ExecIsInsertTrigger.
- ExecIsQuotedIdentOn.
- ExecIsStartup.
- ExecIsTriggerDisabled.
- ExecIsUpdateTrigger.
- IsCheckCnst.
- IsConstraint.
- IsDefault.
- IsDefaultCnst.
- IsExecuted.
- IsExtendedProc.
- IsForeignKey.
- IsMSShipped.
- IsPrimaryKey.
- IsProcedure.
- IsReplProc.
- IsRule.
- IsSystemTable.
- IsTable.
- IsTrigger.
- IsUniqueCnst.
- IsUserTable.
- IsView.
- OwnerId.
- TableDeleteTrigger.
- TableDeleteTriggerCount.
- TableFulltextCatalogId.
- TableFulltextKeyColumn.
- TableHasActiveFulltextIndex.
- TableHasCheckCnst.
- TableHasClustIndex.
- TableHasDefaultCnst.
- TableHasDeleteTrigger.
- TableHasForeignKey.
- TableHasForeignRef.
- TableHasIdentity.
- TableHasIndex.
- TableHasInsertTrigger.

- TableHasNonclustIndex.
- TableHasPrimaryKey.
- TableHasRowGuidCol.
- TableHasTextImage.
- TableHasTimestamp.
- TableHasUniqueCnst.
- TableHasUpdateTrigger.
- TableInsertTrigger.
- TableInsertTriggerCount.
- TableIsFake.
- TableIsPinned.
- TableUpdateTrigger.
- TableUpdateTriggerCount.
- TriggerDeleteOrder.
- TriggerInsertOrder.
- TriggerUpdateOrder.

Функция OBJECTPROPERTY возвращает значение 1 (которое соответствует значению True), 0 (False) или NULL (если задан недопустимый параметр). Исключение составляют параметры, перечисленные ниже.

- OwnerId. При использовании этого параметра функция возвращает идентификатор пользователя базы данных, который является владельцем объекта.
- TableDeleteTrigger, TableDeleteTriggerCount, TableInsertTrigger, TableInsertTriggerCount, TableUpdateTrigger, TableUpdateTriggerCount. При использовании любого из этих параметров функция возвращает идентификатор первого триггера указанного типа.
- TableFulltextCatalogId и TableFulltextKeyColumn. При использовании любого из этих параметров функция возвращает идентификатор полнотекстового каталога; значение 0, которое указывает, что таблица не имеет полнотекстового индекса; или NULL-значение, которое свидетельствует о том, что задан недопустимый параметр.

Функция SQL_VARIANT_PROPERTY

Функция SQL_VARIANT_PROPERTY является очень мощной и возвращает информацию о свойствах переменной типа sql_variant. Эта информация может относиться к таким свойствам, как BaseType, Precision, Scale, TotalBytes, Collation и MaxLength. Для вызова этой функции применяется следующий синтаксис:

```
SQL_VARIANT_PROPERTY (<expression>, <property>)
```

Параметр <expression> представляет собой выражение типа sql_variant, а параметр <property> может иметь одно из значений, перечисленных в табл. Б.4.

Таблица Б.4. Допустимые значения параметра <property> функции SQL_VARIANT_PROPERTY

Значение	Описание	Базовый тип возвращаемого значения sql_variant
BaseType	В число типов данных входят: char, int, money, nchar, ntext, numeric, nvarchar, real, smalldatetime, smallint, smallmoney, text, timestamp, tinyint, uniqueidentifier, varbinary, varchar	sysname
Precision	Точность числового базового типа данных: datetime = 23 smalldatetime = 16 float = 53 real = 24 Для данных типа decimal (p,s) и numeric (p,s) это значение равно p money = 19 smallmoney = 10 int = 10 smallint = 5 tinyint = 3 bit = 1 Для всех прочих типов значение точности равно 0	int
Scale	Число цифр справа от десятичной точки в данных базового числового типа: Для данных типа decimal (p,s) и numeric (p,s) это значение равно s Для данных типа money и smallmoney это значение равно 4 datetime = 3 Для всех прочих типов значение точности равно 0	int
TotalBytes	Число байтов, требуемых для хранения метаданных и значения данных. Если это значение окажется больше 900, то операция создания индекса окончится неудачей	int
Collation	Схема упорядочения, применяемая для данного конкретного значения sql_variant	sysname
MaxLength	Максимальная длина данных этого типа в байтах	int

Функция TYPEPROPERTY

Функция TYPEPROPERTY возвращает информацию о типе данных. Для вызова этой функции применяется следующий синтаксис:

```
TYPEPROPERTY(<type>, <property>)
```

Параметр <type> определяет имя типа данных, а параметр <property> определяет свойство типа данных, значение которого должно быть определено; параметр <property> может принимать одно из значений, перечисленных ниже.

- Precision. Количество цифровых и (или) символьных знаков.
- Scale. Количество десятичных позиций.

- AllowsNull. Принимает значение 1 (которое соответствует значению True) или 0 (False).
- UsesAnsiTrim. Принимает значение 1 (которое соответствует значению True) или 0 (False).

Функции для работы с наборами строк

Функции для работы с наборами строк возвращают объект, которые могут использоваться вместо ссылки на таблицу в операторе T-SQL. Функции, применяемые для работы с наборами строк, перечислены ниже.

- CONTAINSTABLE.
- FREETEXTTABLE.
- OPENDATASOURCE.
- OPENQUERY.
- OPENROWSET.
- OPENXML.

Функция CONTAINSTABLE

Функция CONTAINSTABLE используется в полнотекстовых запросах. Для вызова этой функции применяется следующий синтаксис:

```
CONTAINSTABLE (<table>, {<column> | *}, '<contains_search_condition>')
```

Функция FREETEXTTABLE

Функция FREETEXTTABLE используется в полнотекстовых запросах. Для вызова этой функции применяется следующий синтаксис:

```
FREETEXTTABLE (<table>, {<column> | *}, '<freetext_string>')
```

Функция OPENDATASOURCE

Функция OPENDATASOURCE предоставляет требуемую информацию о соединении. Для вызова этой функции применяется следующий синтаксис:

```
OPENDATASOURCE (<provider_name>, <init_string>)
```

Параметр <provider_name> представляет собой имя, зарегистрированное в реестре как идентификатор ProgID провайдера OLE DB, применяемого для доступа к источнику данных. Другой параметр, <init_string>, должен быть знаком программистам, работающим на языке VB, поскольку он обозначает строку инициализации доступа к провайдеру OLE DB. Например, параметр <init_string> может быть задан следующим образом:

```
"User Id=wonderison;Password=JuniorBlues;DataSource=MyServerName"
```

Функция OPENQUERY

Функция OPENQUERY выполняет заданный запрос '<query>' к внешнему источнику данных на указанном сервере <linked_server>. Для вызова этой функции применяется следующий синтаксис:

```
OPENQUERY(<linked_server>, '<query>')
```

Функция OPENROWSET

Функция OPENROWSET обращается к удаленным данным, которые могут быть получены из источника данных OLE DB. Для вызова этой функции применяется следующий синтаксис:

```
OPENROWSET('<provider_name>'
   {
    '<datasource>';'<user_id>';'<password>'
    | '<provider_string>'
   },
   {
      [<catalog.>][<schema.>]<object>
      | '<query>'
   })
```

Параметр <provider_name> — это строка, представляющая дружественное имя OLE DB, которое должно быть задано в полном соответствии с системным реестром. Параметр <data_source> обозначает строку, соответствующую требуемому источнику данных OLE DB. Параметр <user_id> задает соответствующее имя пользователя, которое должно быть передано провайдеру OLE DB. Параметр <password> указывает пароль, который связан с именем пользователя <user_id>.

Параметр <provider_string> обозначает строку соединения, формат которой зависит от используемого провайдера данных. Этот параметр используется вместо комбинации параметров <datasource>, <user_id> и <password>.

Параметр <catalog> определяет имя каталога и (или) базы данных, который содержит требуемый объект. Параметр <schema> задает имя схемы или владельца требуемого объекта. Параметр <object> определяет имя объекта.

Параметр <query> — это строка, которая выполняется провайдером; этот параметр используется вместо комбинации параметров <catalog>, <schema> и <object>.

Функция OPENXML

Функция OPENXML, после передачи ей документа XML в качестве параметра или поиска документа XML и присваивания содержимого документа переменной, позволяет рассматривать структуру документа и осуществлять в нем выборку данных так, как будто документ XML представляет собой таблицу. Для вызова этой функции применяется следующий синтаксис:

```
OPENXML(<idoc_int> [in],<rowpattern> nvarchar[in],[<flags> byte[in]])
[WITH (<SchemaDeclaration> | <TableName>)]
```

Параметр <idoc_int> представляет собой переменную, которая определена с использованием системной хранимой процедуры sp_xml_preparedocument. Параметр

<rowpattern> содержит определение узла. Параметр <flags> задает отображение между документом XML и набором строк, подлежащим возврату в операторе SELECT. Параметр <SchemaDeclaration> определяет схему XML для документа XML; если в базе данных определена таблица, которая соответствует схеме XML, то вместо параметра <SchemaDeclaration> может использоваться параметр <TableName>.

Прежде чем появится возможность использовать документ XML, его необходимо подготовить с использованием системной хранимой процедуры sp_xml_preparedocument.

Дополнительную информацию об использовании функции OPENXML см. в главе 16.

Функции защиты

Функции защиты возвращают информацию о пользователях и ролях. Перечень этих функций приведен ниже.

- HAS_DBACCESS.
- IS_MEMBER.
- IS_SRVROLEMEMBER.
- SUSER_ID.
- SUSER_NAME.
- SUSER_SID.
- USER.
- USER_ID.

Функция HAS_DBACCESS

Функция HAS_DBACCESS используется для определения того, имеет ли зарегистрированный пользователь доступ к используемой базе данных. Возвращаемое значение 1 означает, что пользователь действительно имеет доступ, а возвращаемое значение 0 свидетельствует о том, что права доступа у пользователя отсутствуют. С другой стороны, возвращаемое NULL-значение указывает, что заданное значение '<database_name>' является недопустимым. Для вызова этой функции применяется следующий синтаксис:

```
HAS_DBACCESS ('<database_name>')
```

Функция IS_MEMBER

Функция IS_MEMBER возвращает информацию о том, является ли текущий пользователь членом указанной группы Windows NT (имеет ли указанную роль SQL Server). Для вызова этой функции применяется следующий синтаксис:

```
IS_MEMBER ({'<group>' | '<role>'})
```

Параметр '<group>' определяет имя группы NT; он должен быть представлен в форме domain\group. Параметр '<role>' указывает имя роли SQL Server. Рассматриваемая роль может относиться к категории постоянной роли базы данных или представлять собой роль, определяемую пользователем, но не может быть ролью сервера.

Эта функция возвращает значение 1, если текущий пользователь является членом указанной группы или имеет указанную роль; значение 0, если текущий пользователь не является членом указанной группы или не имеет указанную роль; и NULL-значение, если параметр с указанием группы или роли является недопустимым.

Функция IS_SRVROLEMEMBER

Функция IS_SRVROLEMEMBER возвращает информацию о том, является ли пользователь представителем указанной роли сервера. Для вызова этой функции применяется следующий синтаксис:

```
IS_SRVROLEMEMBER ('<role>' [,'<login>'])
```

Необязательный параметр '<login>' обозначает имя учетной записи, которая должна быть проверена; по умолчанию рассматривается текущий пользователь. Параметр '<role>' определяет роль сервера и должен иметь одно из перечисленных ниже возможных значений.

- sysadmin.
- dbcreator.
- diskadmin.
- processadmin.
- serveradmin.
- setupadmin.
- securityadmin.

Эта функция возвращает значение 1, если указанная учетная запись является представителем указанной роли; значение 0, если указанная учетная запись не является представителем роли; и NULL-значение, если обозначение роли или учетной записи является недопустимым.

Функция SUSER_ID

Функция SUSER_ID возвращает идентификационный номер учетной записи указанного пользователя. Для вызова этой функции применяется следующий синтаксис:

```
SUSER_ID(['<login>'])
```

Параметр '<login>' представляет собой имя учетной записи указанного пользователя. Если имя учетной записи не задано, то по умолчанию в качестве значения этого параметра используется имя учетной записи текущего пользователя.

Системная функция SUSER_ID включена в программное обеспечение в версии SQL Server 2000 только для обеспечения обратной совместимости, поэтому по возможности вместо нее следует использовать функцию SUSER_SID.

Функция SUSER_NAME

Функция SUSER_NAME возвращает имя учетной записи указанного пользователя. Для вызова этой функции применяется следующий синтаксис:

```
SUSER_NAME([<server_user_id>])
```

Параметр <server_user_id> представляет собой числовой идентификатор учетной записи указанного пользователя. Если параметр <server_user_id> не задан, то по умолчанию в качестве значения этого параметра используется числовой идентификатор учетной записи текущего пользователя.

Системная функция SUSER_NAME включена в программное обеспечение в версии SQL Server 2000 только для обеспечения обратной совместимости, поэтому по возможности вместо нее следует использовать функцию SUSER_SNAME.

Функция SUSER_SID

Функция SUSER_SID возвращает идентификационный номер защиты (SID – Security Identification Number) для указанного пользователя. Для вызова этой функции применяется следующий синтаксис:

```
SUSER_SID(['<login>'])
```

Параметр '<login>' содержит имя учетной записи пользователя. Если параметр '<login>' не задан, то по умолчанию в качестве значения этого параметра используется имя учетной записи текущего пользователя.

Функция SUSER_SNAME

Функция SUSER_SNAME возвращает имя учетной записи, соответствующее указанному идентификационному номеру защиты (SID). Для вызова этой функции применяется следующий синтаксис:

```
SUSER_SNAME([<server_user_sid>])
```

Параметр <server_user_sid> представляет собой SID пользователя. Если значение параметра <server_user_sid> не задано, то по умолчанию в качестве значения этого параметра используется идентификационный номер защиты текущего пользователя.

Функция USER

Функция USER позволяет вставлять в таблицу предоставляемое системой значение имени пользователя базы данных для текущего пользователя, если не задано значение, предусмотренное по умолчанию. Для вызова этой функции применяется следующий синтаксис:

```
USER
```

Функция USER_ID

Функция USER_ID возвращает идентификационный номер базы данных указанного пользователя. Для вызова этой функции применяется следующий синтаксис:

```
USER_ID(['<user>'])
```

Параметр '<user>' представляет собой имя пользователя, которое используется для получения идентификационного номера базы данных. Если значение параметра '<user>' не задано, то по умолчанию в качестве значения этого параметра используется имя текущего пользователя.

Строковые функции

Строковые функции выполняют действия со строковыми значениями и возвращают строковые или числовые значения. Доступные строковые функции перечислены ниже.

- ASCII.
- CHAR.
- CHARINDEX.
- DIFFERENCE.
- LEFT.
- LEN.
- LOWER.
- LTRIM.
- NCHAR.
- PATINDEX.
- QUOTENAME.
- REPLACE.
- REPLICATE.
- REVERSE.
- RIGHT.
- RTRIM.
- SOUNDEX.
- SPACE.
- STR.
- STUFF.
- SUBSTRING.
- UNICODE.
- UPPER.

Функция ASCII

Функция ASCII возвращает значение кода ASCII крайнего левого символа в выражении, заданном параметром <character_expression>. Для вызова этой функции применяется следующий синтаксис:

```
ASCII(<character_expression>)
```

Функция CHAR

Функция CHAR преобразовывает в строковое значение код ASCII (заданный в параметре <expression>). Для вызова этой функции применяется следующий синтаксис:

```
CHAR(<expression>)
```

В качестве параметра <expression> может быть задано любое целое число в пределах между 0 и 255.

Функция CHARINDEX

Функция CHARINDEX возвращает начальную позицию подстроки, заданной параметром <expression>, в строке, обозначенной параметром <character_string>. Для вызова этой функции применяется следующий синтаксис:

```
CHARINDEX(<expression>, <character_string> [, <start_location>])
```

Параметр <expression> обозначает искомую строку, а параметр <character_string> — строку, в которой должен быть выполнен поиск (обычно это — результат выборки данных столбца). Параметр <start_location> задает символьную позицию, с которой должен начаться поиск; если он не задан или представляет собой нечто отличное от положительного целого числа, то поиск осуществляется с начала строки <character_string>.

Функция DIFFERENCE

Функция DIFFERENCE возвращает разницу между значениями SOUNDEX двух выражений в виде целого числа. Для вызова этой функции применяется следующий синтаксис:

```
DIFFERENCE(<expression1>, <expression2>)
```

Эта функция возвращает целочисленное значение в пределах между 0 и 4. Если две строки, представленные параметрами <expression1> и <expression2>, звучат одинаково (например, "blue" и "blew"), функция возвращает значение 4. Если звуковое подобие между строками отсутствует, функция возвращает значение 0.

Функция LEFT

Функция LEFT возвращает подстроку, представляющую собой крайнюю левую часть строки, обозначенной параметром <expression>; отсчет символов подстроки начинается слева, с указанной символьной позиции. Для вызова этой функции применяется следующий синтаксис:

```
LEFT(<expression>, <integer>)
```

Параметр <expression> содержит символьные данные, из которых должна быть извлечена подстрока, начальная позиция которой отсчитывается слева. Параметр <integer> определяет отсчитываемую слева символьную позицию, с которой должно начинаться выделение подстроки; он должен представлять собой положительное целое число.

Функция LEN

Функция LEN возвращает данные о количестве символов в выражении, обозначенном параметром <expression>. Для вызова этой функции применяется следующий синтаксис:

```
LEN(<expression>)
```

Функция LOWER

Функция LOWER преобразовывает все символы верхнего регистра в выражении, представленном параметром <expression>, в символы нижнего регистра. Для вызова этой функции применяется следующий синтаксис:

```
LOWER(<expression>)
```

Функция LTRIM

Функция LTRIM удаляет все начальные пробелы из символьного выражения, обозначенного параметром <character_expression>. Для вызова этой функции применяется следующий синтаксис:

```
LTRIM(<character_expression>)
```

Функция NCHAR

Функция NCHAR возвращает символ Unicode, который имеет указанный целочисленный код <integer_code>. Для вызова этой функции применяется следующий синтаксис:

```
NCHAR(<integer_code>)
```

Параметр <integer_code> должен быть положительным целым числом от 0 до 65 535.

Функция PATINDEX

Функция PATINDEX возвращает начальную позицию подстроки, определяемой шаблоном '<%pattern%>', в символьной строке <expression>, или значение 0, если такая подстрока не найдена. Для вызова этой функции применяется следующий синтаксис:

```
PATINDEX('<%pattern%>', <expression>)
```

Параметр '<%pattern%>' задает шаблон, с помощью которого осуществляется поиск подстроки. В шаблоне могут использоваться подстановочные символы, но первым и последним символами в шаблоне должны быть символы %. Параметр <expression> определяет символьные данные, в которых должен быть выполнен поиск подстроки, соответствующей шаблону (обычно эти данные — результат выборки значений столбца).

Функция QUOTENAME

Функция QUOTENAME возвращает строку Unicode с разграничителями, которые вводятся в целях преобразования заданной строки в допустимый идентификатор SQL Server, содержащий разграничители. Для вызова этой функции применяется следующий синтаксис:

```
QUOTENAME('<character_string>'[, '<quote_character>'])
```

Параметр '<character_string>' представляет строку Unicode, а параметр '<quote_character>' задает односимвольную строку, которая будет использоваться в качестве разграничителя. Параметр '<quote_character>' может задавать одинарную кавычку ('), левую квадратную скобку ([), правую квадратную скобку (]) или двойную кавычку ("). По умолчанию предусмотрено использование квадратных скобок.

Функция REPLACE

Функция REPLACE заменяет все вхождения второй указанной строки в первой указанной строке третьей указанной строкой. Для вызова этой функции применяется следующий синтаксис:

```
REPLACE('<string_expression1>', '<string_expression2>', '<string_expressicn3>')
```

Параметр '<string_expression1>' обозначает строковое выражение, в котором осуществляется поиск. Параметр '<string_expression2>' представляет собой строковое выражение, поиск которого должен осуществляться в строковом выражении '<string_expression1>'. Параметр '<string_expression3>' — это строковое выражение, предназначенное для замены всех вхождений строкового выражения '<string_expression2>'.

Функция REPLICATE

Функция REPLICATE повторяет символьное выражение <character_expression> указанное количество раз. Для вызова этой функции применяется следующий синтаксис:

```
REPLICATE(<character_expression>, <integer>)
```

Функция REVERSE

Функция REVERSE возвращает строку, символы которой расположены в обратном порядке по отношению к строке, заданной символьным выражением <character_expression>. Для вызова этой функции применяется следующий синтаксис:

```
REVERSE(<character_expression>)
```

Функция RIGHT

Функция RIGHT возвращает подстроку, представляющую собой крайнюю правую часть строки, обозначенной параметром <character_expression>; отсчет символов подстроки начинается справа, с символьной позиции, указанной целочисленным параметром <integer>. Для вызова этой функции применяется следующий синтаксис:

```
RIGHT(<character_expression>, <integer>)
```

Параметр <integer> должен представлять собой положительное целое число.

Функция RTRIM

Функция RTRIM удаляет все конечные пробелы из символьного выражения, обозначенного параметром <character_expression>. Для вызова этой функции применяется следующий синтаксис:

```
RTRIM(<character_expression>)
```

Функция SOUNDEX

Функция SOUNDEX возвращает четырехсимвольный код (значение SOUNDEX), который может использоваться для оценки подобия двух строк. Для вызова этой функции применяется следующий синтаксис:

```
SOUNDEX(<character_expression>)
```

Функция SPACE

Функция SPACE возвращает строку из повторяющихся пробелов, длина которой задается целочисленным параметром <integer>. Для вызова этой функции применяется следующий синтаксис:

```
SPACE(<integer>)
```

Функция STR

Функция STR преобразовывает числовые данные в символьные данные. Для вызова этой функции применяется следующий синтаксис:

```
STR(<numeric_expression>[, <length>[, <decimal>]])
```

Параметр <numeric_expression> представляет собой числовое выражение с десятичной точкой. Параметр <length> задает общую длину, включая десятичную точку, цифры и пробелы. Параметр <decimal> определяет количество знаков справа от десятичной точки.

Функция STUFF

Функция STUFF предназначена для удаления подстроки символов указанной длины и вставки на ее место другой подстроки. Для вызова этой функции применяется следующий синтаксис:

```
STUFF(<expression>, <start>, <length>, <characters>)
```

Параметр <expression> представляет собой строку символов, в которой одна подстрока должна быть заменена другой. Параметр <start> определяет, с какого места начинается удаление одной подстроки и вставка другой. Параметр <length> задает количество удаляемых символов. Параметр <characters> определяет новую подстроку, которая должна быть вставлена в строку, заданную параметром <expression>.

Функция SUBSTRING

Функция SUBSTRING возвращает подстроку строки, заданной параметром <expression>. Для вызова этой функции применяется следующий синтаксис:

```
SUBSTRING(<expression>, <start>, <length>)
```

Параметр <expression> определяет символьные данные, из которых должна быть взята подстрока, и может представлять собой строку символов, строку байтов, текст или выражение, которое включает таблицу. Параметр <start> — целое число, которое обозначает начало подстроки. Параметр <length> задает длину подстроки.

Функция UNICODE

Функция UNICODE возвращает код набора символов Unicode, соответствующий первому символу в символьном выражении <character_expression>. Для вызова этой функции применяется следующий синтаксис:

```
UNICODE('<character_expression>')
```

Функция UPPER

Функция UPPER преобразовывает все символы нижнего регистра в символьном выражении <character_expression> в символы верхнего регистра. Для вызова этой функции применяется следующий синтаксис:

```
UPPER(<character_expression>)
```

Системные функции

Системные функции могут использоваться для получения информации о значениях, объектах и параметрах настройки SQL Server. Системные функции перечислены ниже.

- APP_NAME.
- CASE.
- CAST и CONVERT.
- COALESCE.
- COLLATIONPROPERTY.
- CURRENT_TIMESTAMP.
- CURRENT_USER.
- DATALENGTH.
- FORMATMESSAGE.
- GETANSINULL.
- HOST_ID.
- HOST_NAME.
- IDENT_CURRENT.
- IDENT_INCR.
- IDENT_SEED.
- IDENTITY.
- ISDATE.
- ISNULL.
- ISNUMERIC.
- NEWID.
- NULLIF.
- PARSENAME.
- PERMISSIONS.

- ROWCOUNT_BIG.
- SCOPE_IDENTITY.
- SERVERPROPERTY.
- SESSION_USER.
- SESSIONPROPERTY.
- STATS_DATE.
- SYSTEM_USER.
- USER_NAME.

Функция APP_NAME

Функция APP_NAME возвращает имя приложения для текущего сеанса, если это имя задано в приложении в виде данных типа nvarchar. Для вызова этой функции применяется следующий синтаксис:

```
APP_NAME()
```

Функция CASE

Функция CASE используется для проверки списка условий и получения одного из нескольких возможных результатов. Эта функция имеет два возможных описанных ниже формата вызова

- Простая функция CASE применяется для сравнения заданного выражения с множеством простых выражений для определения результата.
- В поисковой функции CASE для определения результата применяется множество булевых выражений.

Оба формата вызова обеспечивают поддержку конструкции ELSE.

Простая функция CASE

Для вызова простой функции CASE используется следующий синтаксис:

```
CASE <input_expression>
   WHEN <when_expression> THEN <result_expression>
   ELSE <else_result_expression>
END
```

Поисковая функция CASE

Для вызова поисковой функции CASE используется следующий синтаксис:

```
CASE
    WHEN <Boolean_expression> THEN <result_expression>
    ELSE <else_result_expression>
END
```

Функции CAST и CONVERT

Функции CAST и CONVERT предоставляют одинаковые функциональные возможности, поскольку обеспечивают преобразование данных из одного типа в другой.

Использование функции CAST

Для вызова функции CAST применяется следующий синтаксис:

```
CAST(<expression> AS <data_type>)
```

Использование функции CONVERT

Для вызова функции CONVERT применяется следующий синтаксис:

```
CONVERT (<data_type>[(<length>)], <expression> [, <style>])
```

Параметр <style> обозначает стиль формата даты, применяемый при преобразовании в символьный тип данных.

Функция COALESCE

Функция COALESCE принимает неопределенное количество параметров и выполняет проверку среди этих параметров для определения первого значения, не равного NULL. Для вызова этой функции применяется следующий синтаксис:

```
COALESCE(<expression> [,...n])
```

Если все параметры равны NULL, функция COALESCE возвращает NULL.

Функция COLLATIONPROPERTY

Функция COLLATIONPROPERTY возвращает информацию о свойствах заданной схемы упорядочения. Для вызова этой функции применяется следующий синтаксис:

```
COLLATIONPROPERTY(<collation_name>, <property>)
```

Параметр <collation_name> представляет собой имя применяемой схемы упорядочения, а параметр <property> обозначает свойство схемы упорядочения, которое требуется определить. Свойство схемы упорядочения может принимать одно из трех значений, приведенных в табл. Б.5.

Таблица Б.5. Свойства схемы упорядочения

Имя свойства	Описание
CodePage	Кодовая страница схемы упорядочения, отличная от Unicode
LCID	Схема упорядочения LCID операционной системы Windows. Если используется схема упорядочения SQL, это свойство равно NULL
ComparisonStyle	Схема упорядочения предусматривает применение стиля сравнения Windows. Если используется схема упорядочения по двоичным значениям или схема упорядочения SQL, это свойство равно NULL

Функция CURRENT_TIMESTAMP

Функция CURRENT_TIMESTAMP применяется исключительно для получения значения текущей даты и времени, представленного как данные типа datetime. Вызов этой функции эквивалентен вызову функции GETDATE(). Для вызова функции CURRENT_TIMESTAMP применяется следующий синтаксис:

```
CURRENT_TIMESTAMP
```

Функция CURRENT_USER

Функция CURRENT_USER применяется исключительно для получения значения идентификатора текущего пользователя, представленного как данные типа sysname. Вызов этой функции эквивалентен вызову функции USER_NAME(). Для вызова функции CURRENT_USER применяется следующий синтаксис:

```
CURRENT_USER
```

Функция DATALENGTH

Функция DATALENGTH возвращает информацию о количестве байтов, используемых для представления выражения, заданного параметром <expression>, в виде целого числа. Эта функция является особенно удобной для работы с типами данных varchar, varbinary, text, image, nvarchar и ntext, которые позволяют хранить данные переменной длины. Для вызова функции DATALENGTH применяется следующий синтаксис:

```
DATALENGTH(<expression>)
```

Функция FORMATMESSAGE

Функция FORMATMESSAGE позволяет использовать сообщения, заданные в таблице sysmessages, чтобы создать новое сообщение. Для вызова этой функции применяется следующий синтаксис:

```
FORMATMESSAGE(<msg_number>, <param_value>[,...n])
```

Параметр <msg_number> представляет собой идентификатор сообщения в таблице sysmessages.

Функция FORMATMESSAGE выполняет поиск сообщения на национальном языке текущего пользователя. Если локализованная версия сообщения отсутствует, то используется версия на американском диалекте английского языка.

Функция GETANSINULL

Функция GETANSINULL позволяет определить заданное по умолчанию значение свойства поддержки NULL для базы данных в виде целого числа. Для вызова этой функции применяется следующий синтаксис:

```
GETANSINULL(['<database>'])
```

Параметр database обозначает имя базы данных, для которой должна быть получена информация о поддержке NULL.

Если заданный параметр поддержки NULL для указанной базы данных допускает применение NULL-значений, а параметр поддержки NULL для столбца или типа данных явно не задан, то функция GETANSINULL возвращает значение 1. Это значение показывает, что применяются предусмотренные по умолчанию стандартом ANSI средства поддержки NULL.

Функция HOST_ID

Функция HOST_ID возвращает идентификатор рабочей станции. Для вызова этой функции применяется следующий синтаксис:

```
HOST_ID()
```

Функция HOST_NAME

Функция HOST_NAME возвращает имя рабочей станции. Для вызова этой функции применяется следующий синтаксис:

```
HOST_NAME()
```

Функция IDENT_CURRENT

Функция IDENT_CURRENT позволяет определить последнее идентификационное значение, созданное для указанной таблицы, в любом сеансе или в любом операторе, применяемом в данной таблице. Применение этой функции полностью аналогично применению системной переменной @@IDENTITY и функции SCOPE_IDENTITY, но ее отличительной особенностью является отсутствие ограничений на выбор области определения, в которой осуществляется поиск для получения требуемого значения.

Для вызова функции IDENT_CURRENT применяется следующий синтаксис:

```
IDENT_CURRENT('<table_name>')
```

Параметр '<table_name>' обозначает имя таблицы, для которой требуется определить текущее идентификационное значение.

Функция IDENT_INCR

Функция IDENT_INCR возвращает значение приращения, заданного во время создания значения для столбца идентификации в таблице (или представлении), которая имеет столбец с типом данных identity. Для вызова этой функции применяется следующий синтаксис:

```
IDENT_INCR('<table_or_view>')
```

Параметр '<table_or_view>' представляет собой выражение, указывающее имя таблицы (или представления), для которой необходимо определить действительную величину приращения идентификационного значения.

Функция IDENT_SEED

Функция IDENT_SEED возвращает данные о том, какое значение имеет начальное число, заданное во время создания идентификационного значения для столбца идентификации в таблице (или представлении), которая содержит столбец типа identity. Для вызова функции IDENT_SEED применяется следующий синтаксис:

```
IDENT_SEED('<table_or_view>')
```

Параметр '<table_or_view>' представляет собой выражение, указывающее имя таблицы (или представления), для которой необходимо определить начальное число, применяемое во время создания идентификационного значения.

Функция IDENTITY

Функция IDENTITY используется для вставки столбца идентификации в новую таблицу. Эта функция применяется только в операторе SELECT с конструкцией INTO <table>. Для вызова функции IDENTITY применяется следующий синтаксис:

```
IDENTITY(<data_type>[, <seed>, <increment>]) AS <column_name>
```

Параметры функции IDENTITY описаны ниже.

- <data_type>. Тип данных столбца идентификации.
- <seed>. Значение, присваиваемое первой строке в таблице. Каждой последующей строке присваивается следующее идентификационное значение, которое равно последнему значению IDENTITY, которое складывается со значением приращения. Если не задан ни параметр <seed>, ни параметр <increment>, оба эти параметра принимают по умолчанию значение 1.
- <increment>. Приращение, которое складывается с начальным числом для определения идентификационного значения, которое должно быть вставлено в очередную строку таблицы.
- <column_name>. Имя столбца, который должен быть вставлен в новую таблицу.

Функция ISDATE

Функция ISDATE применяется для определения того, представляет ли собой входной параметр <expression> допустимую дату. Для вызова этой функции применяется следующий синтаксис:

```
ISDATE(<expression>)
```

Функция ISNULL

Функция ISNULL проверяет параметр <expression> для определения того, представляет ли он собой NULL-значение, и заменяет его указанным значением <replacement_value>. Для вызова этой функции применяется следующий синтаксис:

```
ISNULL(<check_expression>, <replacement_value>)
```

Функция ISNUMERIC

Функция ISNUMERIC позволяет определить, имеет ли параметр <expression> допустимый числовой тип. Для вызова этой функции применяется следующий синтаксис:

```
ISNUMERIC(<expression>)
```

Функция NEWID

Функция NEWID применяется для создания уникального значения типа unique-identifier. Для вызова этой функции применяется следующий синтаксис:

```
NEWID()
```

Функция NULLIF

Функция NULLIF сравнивает два выражения, заданных параметрами <expression1> и <expression2>, и возвращает NULL-значение. Для вызова этой функции применяется следующий синтаксис:

```
NULLIF(<expression1>, <expression2>)
```

Функция PARSENAME

Функция PARSENAME возвращает указанный компонент имени объекта. Для вызова этой функции применяется следующий синтаксис:

```
PARSENAME('<object_name>', <object_piece>)
```

Параметр '<object_name>' задает имя объекта, компонент которого должен быть определен. Параметр <object_piece> обозначает компонент объекта, который должен быть определен. Параметр <object_piece> принимает одно из перечисленных ниже возможных значений.

- 1. Имя объекта.
- 2. Имя владельца.
- 3. Имя базы данных.
- 4. Имя сервера.

Функция PERMISSIONS

Функция PERMISSIONS возвращает битовое значение, которое указывает, какие права доступа к оператору, объекту или столбцу имеет текущий пользователь. Для вызова этой функции применяется следующий синтаксис:

```
PERMISSIONS([<objectid> [, '<column>']])
```

Параметр <object_id> задает идентификатор объекта. Необязательный параметр '<column>' указывает имя столбца, для которого должна быть получена информация о правах доступа.

Функция ROWCOUNT_BIG

Функция ROWCOUNT_BIG во многом аналогична по своему назначению системной переменной @@ROWCOUNT, поскольку возвращает информацию о количестве строк, обработанных в предыдущем операторе. Но эта функция возвращает значение, имеющее тип данных bigint. Для вызова функции ROWCOUNT_BIG применяется следующий синтаксис:

```
ROWCOUNT_BIG()
```

Функция SCOPE_IDENTITY

Функция SCOPE_IDENTITY возвращает последнее значение, вставленное в столбец идентификации в одной и той же области определения (т.е. в пределах одной и той же хранимой процедуры, триггера, функции или пакета). Эта функция аналогична функции IDENT_CURRENT, описанной выше, но IDENT_CURRENT не ограничивает воз-

можность получения идентификационных значений только одной областью определения.

Функция SCOPE_IDENTITY возвращает данные типа sql_variant. Для вызова этой функции применяется следующий синтаксис:

```
SCOPE_IDENTITY()
```

Функция SERVERPROPERTY

Функция SERVERPROPERTY возвращает информацию об эксплуатируемом сервере. Для вызова этой функции применяется следующий синтаксис:

```
SERVERPROPERTY('<propertyname>')
```

Возможные значения свойств сервера, определяемых параметром '<propertyname>', приведены в табл. Б.6.

Таблица Б.6. Свойства сервера и их описания

Имя свойства	Описание
Collation	Имя схемы упорядочения, применяемой для сервера по умолчанию
Edition	Обозначение версии экземпляра SQL Server, установленного на сервере. При использовании этого параметра функция SERVERPROPERTY возвращает одно из следующих значений в формате nvarchar: 'Desktop Engine' 'Developer Edition' 'Enterprise Edition' 'Enterprise Evaluation Edition' 'Personal Edition' 'Standard Edition'
EngineEdition	Версия машины обработки данных, применяемая в экземпляре SQL Server, установленном на сервере: 1. Personal Engine или Desktop Engine 2. Standard Engine 3. Enterprise Engine (функция SERVERPROPERTY возвращает значение EngineEdition применительно к версиям Enterprise Engine, Enterprise Evaluation Engine и Developer Engine)
InstanceName	Имя экземпляра, к которому подключен пользователь
IsClustered	Информация о том, выполнена ли настройка конфигурации экземпляра сервера для работы в отказоустойчивом кластере. 1. Настройка для работы в кластере выполнена. 0. Настройка для работы в кластере не выполнена. NULL. Недопустимые входные данные или ошибка
IsFullTextInstalled	Информация о том, установлены ли для работы с текущим экземпляром SQL Server средства поддержки полнотекстового поиска. 1. Средства поддержки полнотекстового поиска установлены. 0. Средства поддержки полнотекстового поиска не установлены. NULL. Недопустимые входные данные или ошибка
IsIntegratedSecurityOnly	Информация о том, находится ли сервер в режиме применения интегрированных средств обеспечения безопасности. 1. Применяются интегрированные средства обеспечения безопасности. 0. Интегрированные средства обеспечения безопасности не применяются. NULL. Недопустимые входные данные или ошибка

Окончание табл. Б.6

Имя свойства	Описание
IsSingleUser	Информация о том, предназначена ли применяемая инсталляция сервера для работы в однопользовательском режиме. 1. Применяемая инсталляция сервера предназначена для работы в однопользовательском режиме. 0. Применяемая инсталляция сервера не предназначена для работы в однопользовательском режиме. NULL. Недопустимые входные данные или ошибка
IsSyncWithBackup	Информация о том, является ли база данных опубликованной базой данных или базой данных распространения, и может ли быть восстановлена без нарушения текущего процесса транзакционной репликации. 1. Значение True. 0. Значение False
LicenseType	Информация о типе лицензии, установленной для рассматриваемого экземпляра SQL Server. PER_SEAT. Режим "per seat" ("на каждое рабочее место"). PER_PROCESSOR. Режим "per processor" ("на каждый процессор"). DISABLED. Лицензирование отменено
MachineName	Имя компьютера с операционной системой Windows, на котором эксплуатируется экземпляр сервера. Если экземпляр эксплуатируется в кластере (представляет собой экземпляр SQL Server, эксплуатируемый на виртуальном сервере под управлением системы Microsoft Cluster Server), то функция SERVERPROPERTY возвращает имя виртуального сервера
NumLicenses	Число лицензий на рабочие места, зарегистрированных для рассматриваемого экземпляра SQL Server, если он эксплуатируется в режиме применения отдельной лицензии для каждого рабочего места (в режиме "per seat"). Число лицензий на процессоры, зарегистрированных для рассматриваемого экземпляра SQL Server, если он эксплуатируется в режиме применения отдельной лицензии для каждого процессора (в режиме "per processor")
ProcessID	Идентификатор процесса службы SQL Server (значение ProcessID позволяет определить, какой экземпляр программы sqlservr.exe принадлежит данному экземпляру сервера)
ProductVersion	Сведения о версии экземпляра SQL Server, представленные в формате, который полностью аналогичен формату 'major.minor.build', применяемому для обозначения проектов Visual Basic ("<основной номер версии>.<дополнительный номер версии>.<номер выпуска>")
ProductLevel	Обозначение версии эксплуатируемого в настоящее время экземпляра SQL Server. 'RTM'. Поставляемая версия. 'SPn'. Версия со служебным пакетом. 'Bn'. Бета-версия
ServerName	Информация о сервере Windows NT и экземпляре, относящаяся к указанному экземпляру SQL Server

Функция SERVERPROPERTY широко применяется в корпорациях, подразделения которых расположены на нескольких площадках, поскольку разработчики должны иметь информацию о каждом сервере.

Функция SESSION_USER

Функция SESSION_USER позволяет вставить в таблицу предоставляемое системой значение имени пользователя текущего сеанса, если не задано значение, применяемое по умолчанию. Для вызова этой функции применяется следующий синтаксис:

```
SESSION_USER
```

Функция SESSIONPROPERTY

Функция SESSIONPROPERTY позволяет определить опции SET, применяемые в текущем сеансе. Для вызова этой функции применяется следующий синтаксис:

```
SESSIONPROPERTY (<option>)
```

Эта функция предназначена для использования в тех условиях, когда эксплуатируются хранимые процедуры, которые изменяют свойства сеанса в определенных ситуациях. Вообще говоря, эту функцию не приходится использовать слишком часто, поскольку на этапе прогона, как правило, не должны происходить значительные изменения многочисленных опций SET.

Функция STATS_DATE

Функция STATS_DATE возвращает дату последнего обновления статистических данных, относящихся к указанному индексу. Для вызова этой функции применяется следующий синтаксис:

```
STATS_DATE(<table_id>, <index_id>)
```

Функция SYSTEM_USER

Функция SESSION_USER позволяет вставить в таблицу предоставляемое системой значение имени пользователя, который в настоящее время работает в системе, если не задано значение, применяемое по умолчанию. Для вызова этой функции применяется следующий синтаксис:

```
SYSTEM_USER
```

Функция USER_NAME

Функция USER_NAME возвращает имя пользователя базы данных. Для вызова этой функции применяется следующий синтаксис:

```
USER_NAME([<id>])
```

Параметр <id> определяет идентификационный номер пользователя, имя которого должно быть определено с помощью функции; если значение этого параметра не задано, применяется идентификационный номер текущего пользователя.

Функции для работы с текстом и изображениями

Функции для работы с текстом и изображениями позволяют выполнять операции с текстом или с данными изображений. Перечень этих функций приведен ниже.

- PATINDEX (эта функция описана в разделе "Строковые функции").
- TEXTPTR.
- TEXTVALID.

Функция TEXTPTR

Функция TEXTPTR проверяет значение текстового указателя, который соответствует столбцу типа text, ntext или image и возвращает значение varbinary. Перед вызовом на выполнение операторов READTEXT, WRITETEXT и UPDATE необходимо проверить этот текстовый указатель, чтобы убедиться в том, что он указывает на первую текстовую страницу. Для вызова этой функции применяется следующий синтаксис:

```
TEXTPTR(<column>)
```

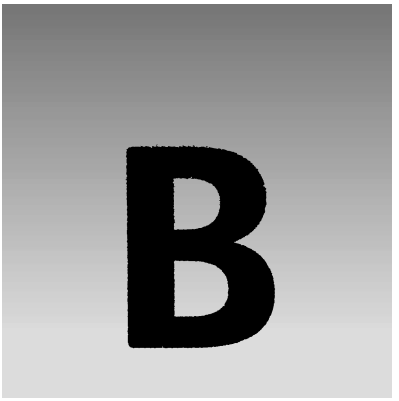
Функция TEXTVALID

Функция TEXTPTR позволяет проверить, является ли указанный текстовый указатель допустимым. Для вызова этой функции применяется следующий синтаксис:

```
TEXTVALID('<table.column>', <text_ptr>)
```

Параметр '<table.column>' задает имя используемой таблицы и столбца. Параметр <text_ptr> определяет текстовый указатель, подлежащий проверке.

Эта функция возвращает значение 0, если указатель является недопустимым, а если указатель является допустимым, возвращает значение 1.



Выбор подходящего инструментального средства

Настоящее приложение содержит рекомендации по выбору наиболее подходящих инструментальных средств формирования ER-диаграмм, разработки кода и резервного копирования. Во всех своих книгах, посвященных описанию СУБД SQL Server, автор выносит данную тему в приложение по той причине, что рассматриваемое здесь программное обеспечение не входит в основной состав программ SQL Server.

Сведения, приведенные в данном приложении, в основном объективно отражают современное состояние дел в указанной области программного обеспечения, но не направлены на то, чтобы склонить читателя к приобретению какого-либо конкретного программного продукта.

Общей целью данного приложения является описание некоторых доступных инструментальных средств и их характеристик, что позволяет получить общее представление о состоянии дел в этой области. Эта информация поможет читателю выбрать наиболее подходящие программы для работы с СУБД SQL Server.

Основные категории программ, рассматриваемых в данном приложении, перечислены ниже.

- Инструментальные средства подготовки ER-диаграмм.
- Инструментальные средства разработки кода.
- Программы резервного копирования.

Инструментальные средства подготовки ER-диаграммы

Диаграмма "сущность–связь" (Entity Relationship Diagram – ER-диаграмма) представляет собой одно из основных средств изучения и проектирования баз данных. При разработке баз данных производственного назначения буквально невозможно обойтись без использования качественных инструментальных средств подготовки ER-диаграмм.

Безусловно, программные средства проектирования, основанные на использовании ER-диаграмм, стоят довольно дорого (в частности, при покупке определенного программного продукта, хотя и очень мощного, приходится выплачивать до 15 тысяч долларов в расчете на одно рабочее место), но их приобретение вполне оправдано. Основные возможности программных средств формирования ER-диаграмм описаны в следующих разделах.

Логическое и физическое проектирование

Во всех основных программных продуктах, основанных на использовании ER-диаграмм, соблюдается принцип разделения логического и физического проектирования. Следует отметить, что логическое моделирование, по-видимому, не нашло должного отражения в данной книге, поскольку в ней главным образом рассматривались программные продукты, которые входят в состав программного обеспечения SQL Server, а они просто не предназначены для использования в логическом проектировании. Но по мере того, как специалист по базам данных начинает все чаще сталкиваться с проблемами реализации проекта, задача разделения логического и физического проектирования становится для него все более важной.

Ниже приведено краткое описание различий между логической и физической моделями, которые применяются при логическом и физическом проектировании.

Логическая модель предназначена для создания общего представления связей между сущностями (абстрактными представлениями данных), применяемыми при проектировании структуры базы данных. В логической модели исключены многие подробности, поэтому она позволяет в большей степени сосредоточиться на тех сущностях и связях, которые являются предметом моделирования, и меньше внимания уделять реализации разрабатываемой модели. Создание и сопровождение логической модели позволяет понять, какой вид в конечном итоге будет иметь готовая база данных, не погружаясь при этом в ненужные подробности. Кроме того, логическая модель может использоваться для представления источников данных, внешних по отношению к базе данных (например, с помощью логической модели можно указать, что доступ к базе данных номеров кредитных карточек будет осуществляться через некоторую службу, но в собственной базе данных номера кредитных карточек храниться не будут).

С другой стороны, физическая модель представляет полную спецификацию базы данных, предназначенной для эксплуатации. Физическая модель предназначена исключительно для описания действительно применяемых таблиц и представлений, а также для определения связей между этими объектами базы данных.

Таким образом, основными структурными единицами в логической модели являются сущности, а в физической модели – таблицы. Единственной сущности логической

модели может соответствовать несколько таблиц физической модели. Например, для представления сущности Order (Заказ), по-видимому, необходимо предусмотреть в физической базе данных и таблицу заголовков (которая содержит определения атрибутов, общих для всех заказов), и таблицу расшифровок (содержащую данные об отдельных элементах заказов). Аналогичным образом, логическая модель позволяет легко представить связь "многие ко многим" в виде сущности, отображающей непосредственные связи, а в физической модели приходится использовать соединительную таблицу, которая представляет связи "один ко многим" с каждой из двух родительских таблиц (современные реляционные СУБД не позволяют непосредственно представить связи "многие ко многим").

Для успешного решения задач создания логической модели и ее преобразования в физическую модель для дальнейшей реализации применяются инструментальные средства разработки ER-диаграмм. Эти инструментальные средства должны обеспечивать представление в логической модели таких сущностей и связей, которые не всегда находят свое отражение в физической модели, но применяются в процессе эксплуатации базы данных. К примерам подобных сущностей и связей относятся компоненты запросов, применяемых на практике вместо хранимых процедур, и рабочие таблицы, которые существуют только в оперативной памяти (например, кэш с данными, изменения в которых отслеживаются применяемыми программными компонентами). Логическая модель должна воплощать в себе все необходимые структуры данных и вместе с тем обеспечивать удобство разделения структур данных, реализуемых и не реализуемых в физической базе данных. Кроме того, логическая модель должна способствовать максимальному упрощению задачи перемещения в физическую модель тех структур данных, которые в конечном итоге должны найти свое место в фактически создаваемой базе данных.

Основные задачи создания сценариев

Практически все инструментальные средства формирования ER-диаграмм позволяют подготавливать сценарии создания базы данных на основе результатов разработки схематической структуры данных. По-видимому, одной из наиболее важных особенностей этих средств является также то, что большинство из них предоставляет возможность создавать сценарии для нескольких разных платформы. Это означает, что разработанная единожды диаграмма "сущность–связь" может применяться при выработке сценариев для различных СУБД. В частности, почти во всех инструментальных средствах предусмотрена поддержка SQL Server и Oracle. Кроме того, часто можно встретить также такие программы, которые обеспечивают поддержку DB/2, Access, MySQL и других СУБД. Применение подобных инструментальных средств позволяет существенно сократить затраты времени в тех случаях, когда необходимо организовать эксплуатацию баз данных на нескольких разных платформах.

Обратное проектирование

Итак, многие инструментальные средства позволяют создавать ER-диаграммы и формировать на их основе базу данных, но иногда требуется решить обратную задачу – автоматически анализировать структуру базы данных. Применяемые для этой цели инструментальные средства должны обеспечивать возможность выполнять **обратное проектирование** базы данных. Процесс обратного проектирования состоит в том, что с помощью соответствующей программы осуществляется подключение к

базе данных, сканирование базы данных и формирование диаграммы, в которой правильно отображаются все таблицы, представления, триггеры, ограничения целостности, индексы и связи.

Итак, программы создания ER-диаграммы должны также предоставлять возможность осуществления обратного проектирования базы данных, поскольку в этом действительно часто возникает необходимость. Поэтому лучше приобрести инструментальное средство, позволяющее решать обе эти задачи, а не тратить деньги на покупку двух программ — для создания ER-диаграмм и обратного проектирования.

К счастью, большинство инструментальных средств высокого класса, которые рассматриваются в настоящем приложении, позволяют решать обе указанные задачи и действительно обеспечивают получение очень качественных результатов. А некоторые из них позволяют даже импортировать из базы данных хранимые процедуры.

Синхронизация

Одним из тех аспектов эксплуатации базы данных, из-за которых чаще всего возникают проблемы, являются любые изменения, внесенные после создания физической базы данных. Эти проблемы становятся еще более существенными после того, как начинается загрузка в базу данных реальных данных. Иными словами, отслеживание изменений, вносимых в базу данных, обычно становится одним из тех направлений работы, которому всегда приходится уделять внимание.

Значительную сложность может представлять даже сам контроль над тем, какие изменения были внесены в модели и действительно осуществлены в самой базе данных, поскольку такие упущения, когда задуманная модификация базы данных так и остается нереализованной, можно допустить даже очень легко. Инструментальные средства формирования ER-диаграмм, обеспечивающие синхронизацию, позволяют прежде всего устранить расхождения между моделью и физической базой данных.

Программные продукты, позволяющие проанализировать физическую базу данных, сравнить ее с моделью, а затем составить список несоответствий, предоставляют многие из основных поставщиков инструментальных средств создания ER-диаграмм. Применение подобного программного обеспечения позволяет существенно упростить работу, если в процессе эксплуатации базы данных постоянно приходится вносить в нее не только мелкие, но и крупные изменения (причем иногда мелкие изменения являются наиболее опасными, поскольку о них вскоре забывают). Еще одним удобным способом использования функциональных возможностей, предоставляемых таким программным обеспечением, является сопровождение системы, в которой предусмотрен удаленный доступ для обслуживания в нерабочее время. В этом случае возможно провести синхронизацию и отразить все внесенные изменения на ER-диаграмме. Но перспективы использования указанных инструментальных средств этим не ограничиваются.

Прежде всего следует отметить, что после того как обнаруживаются различия между проектом и реальной базой данных (которые должны быть отражены на ER-диаграмме), возникает необходимость в написании кода, позволяющего внести требуемые изменения.

На первый взгляд кажется, что задача написания кода, предназначенного для внесения изменений в базу данных, не является такой уж трудной. И действительно, если требуется лишь добавить еще один столбец в конце таблицы, в этом нет ничего сложного. Но достаточно представить себе, что столбец должен быть добавлен не по-

сле других столбцов, а между другими столбцами таблицы. Для осуществления этой операции необходимо выполнить описанные ниже действия.

1. Создать таблицу с другим именем, имеющую требуемую компоновку.
2. Скопировать все данные в соответствующие столбцы новой таблицы.
3. Удалить старую таблицу.
4. Переименовать новую таблицу.

Очевидно, что количество требуемых действий невелико, но реализовать их на практике довольно сложно. А теперь рассмотрим, как указанные действия могут быть осуществлены в реально эксплуатируемой базе данных. Достаточно представить себе, что на старую таблицу ссылались какие-либо внешние ключи. Это означает, что связь, в которой заданы эти внешние ключи, должна быть уничтожена, поскольку иначе нельзя будет удалить старую таблицу (выполнить шаг 3 из приведенного выше списка). После этого необходимо снова определить связь, заданную с помощью внешнего ключа (и обновить статистические данные). Итак, рассмотрим, какие действия должны быть выполнены в указанных условиях.

1. Создать таблицу с другим именем, имеющую требуемую компоновку.
2. Скопировать все данные в соответствующие столбцы новой таблицы.
3. Удалить связь, в которой заданы внешние ключи, из той таблицы, которая ссылается на старую таблицу.
4. Удалить старую таблицу.
5. Переименовать новую таблицу.
6. Снова сформировать для всех таблиц ограничения внешних ключей, которые были только что удалены.

Вполне очевидно, что наличие даже одного дополнительного условия приводит к существенному усложнению рассматриваемой задачи.

Итак, рассмотрим, какие программные средства могут применяться для осуществления сложных преобразований таблиц. Некоторые основные инструментальные средства создания ER-диаграмм (но не все) позволяют составлять сценарии для решения указанной задачи. После завершения этапа сравнения процедуры синхронизации такие инструментальные средства предоставляют пользователю диалоговое окно, позволяющее провести сравнение. В одной части этого диалогового окна находятся определения, заданные в ER-диаграмме, а в другой — определения, реализованные в базе данных. При этом пользователь имеет возможность принимать решения, касающиеся того, какой стороне должно быть отдано предпочтение при устранении каждого расхождения, — должно ли быть распространено на базу данных изменение, предусмотренное в ER-диаграмме, или следует принять как требующее отображения на ER-диаграмме то изменение, которое внесено в базу данных.

Очевидно, что подобные инструментальные средства позволяют сэкономить немало часов работы.

Следует учитывать, что нельзя полностью рассчитывать на отсутствие каких-либо ошибок в сценариях внесения изменений, составленных с помощью указанных инструментальных средств. Поэтому, прежде чем применить к эксплуатируемой базе данных какой-либо сценарий обновления, обязательно проведите всестороннюю проверку каждого сценария, сформированного инструментальным средством создания ER-диаграмм, на испытательной базе данных и только после этого вносите изменения в реальные данные.

Макрокоманды

Некоторые инструментальные средства предоставляют возможность формировать на основе ER-диаграмм макрокоманды. Макрокоманды могут использоваться для упрощения повторяющихся заданий, а также позволяют автоматизировать создание хранимых процедур и триггеров, автоматически подставляя информацию об именах и типах данных в специально созданные шаблоны триггеров и хранимых процедур.

Но самым значительным преимуществом подхода к разработке с использованием макрокоманд является достигаемая при этом возможность легко вносить изменения — достаточно просто модифицировать таблицу, и после этого происходит автоматическое внесение изменений в хранимые процедуры и триггеры (но для этого необходимо хорошо знать язык макрокоманд).

Недостатком подхода к разработке, основанного на использовании макрокоманд, является потребность в дополнительном обучении. Чтобы хорошо освоить язык макрокоманд, необходимо затратить значительное время (к тому же для разных инструментальных средств определены разные языки, которые чаще всего связаны с объектной моделью, которая используется для представления объектов базы данных). Кроме того, язык макрокоманд приходится изучать в полном объеме, поскольку невозможно ограничиваться только теми аспектами его применения, которые интересуют вас в данный момент.

Интеграция с другими инструментальными средствами (автоматическое формирование кода)

Некоторые версии инструментальных средств создания ER-диаграмм обладают даже собственными возможностями формирования кода или обеспечивают интеграцию для этой цели с другими инструментальными средствами. Наличие таких возможностей может оказаться действительно полезным во многих отношениях, прежде всего с точки зрения создания прототипов и интеграции с логической моделью.

Например, предположим, что имеется логическая модель, предусматривающая вызов нескольких объектов, отличных от объектов базы данных. Подобные инструментальные средства позволяют автоматически вырабатывать шаблонный код для таких объектов. Например, может быть создан код для доступа к свойствам и создания заглушек, обеспечивающих вызов методов. Это позволяет избавиться от огромного объема достаточно трудоемкой работы.

Как и при использовании программных средств, которые описаны в предыдущем разделе, посвященном синхронизации, применяя инструментальные средства автоматического формирования кода, необходимо соблюдать осторожность. Не следует думать, что весь код, сформированный с их помощью, является правильным; обязательно проверяйте сгенерированный код, чтобы убедиться в том, что его применение действительно позволяет достичь намеченной цели. Но несмотря на дополнительные затраты времени, связанные с необходимостью просмотра выработанного кода, использование генераторов кода для формирования объектов базы данных способствует значительному повышению производительности труда программистов.

Другие категории инструментальных средств

Ниже перечислены некоторые другие области применения инструментальных средств, заслуживающие внимания разработчика.

- **Автоматическая загрузка справочных данных**. В большинстве баз данных предусмотрено насколько справочных таблиц. Во время подготовки базы данных к работе должна быть выполнена загрузка в эти таблицы данных, которые остаются неизменными (в качестве распространенных примеров таких таблиц можно указать таблицы с перечнем штатов США или таблицы с названиями государств). Проблема заключается в том, что после каждого повторного восстановления базы данных приходится загружать справочные данные. Некоторые инструментальные средства, относящиеся к этой категории, позволяют встраивать сценарии "предварительной загрузки" непосредственно в ER-диаграмму. Это дает возможность каждый раз при создании базы данных осуществлять предварительную загрузку таблиц с необходимыми данными.
- **Вырезка и вставка**. Некоторые используемые инструментальные средства создания ER-диаграмм не поддерживают операции связывания и внедрения. Это означает, что отсутствует возможность осуществлять вырезку данных непосредственно в окне инструментального средства и последующую вставку этих данных в окно программы Word или какого-то другого текстового процессора. Безусловно, отсутствие указанной возможности создает огромные сложности, когда возникает необходимость подготовить документацию по базе данных.
- **Поддержка методологий создания диаграмм**. Практика показывает, что основная часть инструментальных средств создания ER-диаграмм поддерживает две наиболее распространенные методологии – IDEF1X и IE. А основные различия между инструментальными средствами заключаются в том, что они обеспечивают разную степень поддержки указанных методологий. Например, по меньшей мере одно из основных инструментальных средств обеспечивает возможность применения методологии IE для физической базы данных, но не допускает ее использование для логической базы данных (вместо этого пользователь вынужден применять предусмотренную в этой программе собственную методологию). Если для разработчика такие нюансы не имеют особого значения, то не о чем беспокоиться, но по крайней мере необходимо знать, с чем вы работаете.
- **Предметные области**. Некоторые базы данных представляют такое количество отдельных сущностей, которое нельзя назвать иначе, чем колоссальным. В результате этого практически исключается возможность сопровождать ER-диаграмму с помощью окна, развернутого на дисплее компьютера (поскольку обычно применяются дисплеи с диагональю не больше 21 дюйма, да и то обладатели таких дисплеев могут считать себя счастливцами). Один из способов преодоления указанной сложности, применяемых в инструментальных средствах, состоит в определении предметных областей. Использование предметных областей позволяет по существу выбирать по критерию то, что должно появиться на основной схеме. Таблицы, которые необходимо отнести к той или другой предметной области, выбирает сам разработчик. Благодаря этому появляется возможность управлять различными частями базы данных, разбивая ее на меньшие секции.

- **Интеграция со средствами управления исходным кодом**. В настоящее время интеграцию с утилитами управления исходным кодом (такими как Source Safe или PVCS) на должном уровне обеспечивают только инструментальные средства самого высокого класса (которые достигают стоимости в 10 тысяч долларов за одно рабочее место). Автор уже давно рассчитывает на то, что такое положение изменится, но этого пока не происходит. Тем не менее надежда еще остается.

Наиболее важной особенностью инструментальных средств создания ER-диаграмм является то, что в них применяются разные форматы представления ER-диаграмм, поэтому отсутствует возможность сравнивать варианты ER-диаграмм для определения того, какие фактически отличия в них имеются.

Примеры инструментальных средств

Ниже перечислены некоторые наиболее широко распространенные программные продукты, применяемые для создания ER-диаграмм.

- ErWin. Программа ErWin пользуется наибольшей популярностью. Какое-то время эта программа считалась буквально эталоном программного обеспечения данного типа и сохраняла монопольное положение на рынке. Безусловно, в наши дни программа ErWin сталкивается с большей конкуренцией, но продолжает оставаться на рынке инструментальных средств создания ER-диаграмм одним из выдающихся программных продуктов, относящихся к категории программ высшего качества.

- ER Studio. Фактически программа ER Studio впервые смогла потеснить программу ErWin на широком рынке. Безусловно, другие программные продукты, начиная с определенного времени, составляли конкуренцию программе ErWin, но лишь разработчикам ER Studio впервые удалось существенно изменить подход к трактовке некоторых функций инструментальных средств создания ER-диаграмм. Эти нововведения не остались без внимания разработчиков других программ, и поэтому в настоящее время программа ER Studio, безусловно, утратила те отличительные особенности, которые выделяли ее среди других программных продуктов этой категории, но осталась среди лидеров.

- PowerDesigner. Программа PowerDesigner также относится к числу давно известных инструментальных средств данной категории. Одной из отличительных особенностей этой программы является то, что она весьма успешно позволяет разделять аспекты создания логической и физической моделей.

- Visio. Программа Visio не обладает некоторыми из наиболее сложных функций инструментальных средств, перечисленных выше, но вместе с тем не имеет столь высокую стоимость. В программе Visio предусмотрены способы представления большинства из наиболее распространенных методологий создания диаграмм, но она имеет недостатки в части межплатформенной поддержки, а также не обладает всеми возможностями формирования сценариев.

Инструментальные средства подготовки кода

Различные версии интерактивной среды разработки для C#, C++, Visual Basic, Java, а также для многих других современных языков программирования стали эталоном исключительно надежной среды разработки. В настоящее время невозможно представить себе инструментальные средства разработки для одного из ведущих языков программирования, которые не обеспечивали бы высокой степени интеграции с системой управления исходным кодом и не обладали бы великолепными возможностями отладки. А что касается языка SQL, то здесь сделан только первый шаг в указанном направлении.

Начиная с версии SQL Server 2005 корпорация Microsoft перешла к использованию среды разработки Visual Studio даже для самых простых операторов T-SQL. По-видимому, такой подход должен вполне устраивать разработчика, который использует исключительно язык T-SQL или работает в языковой среде .NET. А если разработчику приходится поддерживать другие платформы, то он может быть вынужден заняться поиском других версий среды разработки на языке SQL, предоставляемых независимыми поставщиками. Но нужно быть готовым к тому, что среда отладки в значительно меньшей степени интегрирована со средой разработки.

Примеры инструментальных средств

Потребность в использовании отдельного программного продукта для редактирования запросов неуклонно снижалась с тех пор, как была выпущена версия SQL Server 7.0, в которой было так много сделано для усовершенствования среды редактирования. Указанная тенденция стала еще более отчетливо выраженной после того, как в программе Visual Studio была улучшена поддержка запросов SQL. Тем не менее следует также назвать еще несколько инструментальных средств, обладающих своими преимуществами (по-видимому, наиболее важным из них является способность подключаться к другим СУБД, а не только к СУБД SQL Server). Некоторые наиболее широко применяемые инструментальные средства перечислены ниже.

- Ultra Edit. Программа Ultra Edit фактически не позволяет даже подключиться к базе данных, поскольку она представляет собой текстовый редактор. Причина, по которой эта программа указана в списке, заключается в том, что она позволяет создать определяемую пользователем библиотеку ключевых слов (что дает возможность применять цветовые обозначения для ключевых слов и тому подобных синтаксических элементов) и вместе с тем предоставляет весьма удобные возможности редактирования текста (которые позволяют очень легко осуществлять глобальный поиск и замену, а также выполнять другие подобные функции текстового редактора).
- SQL Editor. Программа SQL Editor выпущена той же компанией Embarcadero, в которой была разработана программа ER Studio. Программа SQL Editor позволяет использовать запросы, охватывающие несколько платформ.
- TOAD. Программа TOAD создана одним из ведущих изготовителей межплатформенных инструментальных средств администрирования SQL и очень широко известна в сообществе пользователей Oracle. Программа TOAD представляет собой весьма эффективную среду разработки, но с ее применением

связаны все те преимущества и недостатки, которые могут быть свойственны программному обеспечению, рассчитанному на работу только с одним сервером базы данных.

- SQL Compare. Программа SQL Compare представляет собой инструментальное средство, позволяющее сравнивать две базы данных и формировать сценарии внесения изменений, позволяющие устранить различия между этими базами данных.

Утилиты резервного копирования

Безусловно, задача резервного копирования в большей степени относится к сфере деятельности администратора базы данных, а не разработчика, но, по мнению автора, разработчики не должны оставлять без внимания такую тему, как резервное копирование. Дело в том, что резервное копирование в значительной степени выходит за рамки средств предотвращения потери данных в системе, применяемой для разработки. По мнению автора, наиболее важным побудительным мотивом для использования утилит резервного копирования должно быть оказание помощи конечным пользователям, особенно если разрабатываемое приложение будет эксплуатироваться в таких условиях, что его сопровождением не сможет заниматься по-настоящему подготовленный администратор базы данных.

Для СУБД SQL Server предусмотрены собственные варианты резервного копирования, так же как и собственные способы создания диаграмм для базы данных и ввода кода. При подготовке каждого нового выпуска программного обеспечения SQL Server корпорация Microsoft прикладывает определенные усилия для обеспечения того, чтобы процедура резервного копирования стала немного проще по сравнению с предыдущей версией. Тем не менее еще многое должно быть сделано в этом направлении. Кроме того, в случае необходимости всегда можно воспользоваться инструментальными средствами независимых поставщиков.

В некоторых из наиболее важных инструментальных средств резервного копирования предусмотрен более удобный пользовательский интерфейс, чем в программном обеспечении SQL Server. Кроме того, многие из этих инструментальных средств позволяют осуществлять резервное копирование на несколько разных устройств хранения, а также выполнять операции, подобные динамическому сжатию резервной копии в процессе ее записи (заслуживает удивления то, насколько много места экономится за счет этого).

Безусловно, автор считает себя прежде всего разработчиком, поэтому в большей степени готов потратить свои деньги на приобретение других программ, но не утилит резервного копирования. Но я все равно рекомендую читателю ознакомиться с имеющимися программами данной категории, чтобы иметь возможность квалифицированно проконсультировать своих заказчиков.

Примеры инструментальных средств

- SQL Backup. Программа SQL Backup выпущена той же компанией Red Gate, в которой была разработана программа SQL Compare. Это — относительно недорогая утилита управления резервным копированием, выпущенная независимым поставщиком.

- Backup Exec. Программа Backup Exec относится к категории утилит резервного копирования, которая может применяться в широких масштабах, не ограничиваясь только базами данных SQL Server. В комплект программы Backup Exec входят инструментальные средства, позволяющие создать резервную копию всех файлов операционной системы, но в этой программе учитывается также архитектура формирования резервных копий SQL Server и предусмотрена возможность автоматизировать этот процесс, а также вести библиотеку созданных ранее резервных копий.

Резюме

Инструментальные средства для SQL Server применяются в основном по такому же принципу, как и инструментальные средства любых других типов. Речь идет о том, что вы не обязаны пользоваться исключительно таким инструментальным средством, которое в наибольшей степени подходит для выполнения текущего задания, но, безусловно, выбор правильного инструмента оказывает огромное влияние на производительность труда разработчика и отражается на том, насколько удовлетворительными (или разочаровывающими) станут достигнутые результаты и полученный в конечном итоге программный продукт.

Найдите время, чтобы ознакомиться со всеми программными продуктами, имеющимися на рынке. Безусловно, некоторые из этих инструментальных средств могут показаться дорогостоящими, но расцените целесообразность их приобретения с той точки зрения, какая экономия трудозатрат и ресурсов может быть достигнута с их помощью, иными словами, сопоставьте затраты и выгоды. По крайней мере, вашего внимания заслуживает одно или несколько инструментальных средств, перечисленных в этом приложении.

Очень простые примеры обеспечения связи

В настоящей книге приведен огромный объем сведений о том, как эксплуатировать базу данных SQL Server. Разумеется, эти сведения очень важны, если все уже подготовлено для работы в базе данных, но в действительности для обеспечения такой работы должны быть дополнительно осуществлены определенные действия. Как правило, сами разработчики подключаются к базе данных в программе Management Studio и непосредственно выполняют в ней запросы, однако подавляющее большинство пользователей никогда фактически не получают прямой доступ к базе данных; им предоставляется лишь возможность работать с окнами ввода и формирования отчетов в приложении, написанном разработчиком.

С учетом сказанного, по-видимому, имеет смысл разобраться в том, каким образом приложение фактически взаимодействует с базой данных. Исключительно этой теме посвящено очень много книг (к тому же, не считая описания простого соединения, сама тематика обеспечения взаимодействия с базой данных является очень обширной), поэтому в настоящем приложении будут приведены только самые основные сведения. Таким образом, наша задача состоит в том, чтобы информация, приведенная в этом разделе, позволила разобраться в рассматриваемых примерах. В настоящем приложении можно найти простые примеры того, как подключиться к базе данных, выполнить запрос и отключиться с помощью программы на языке C# или VB.NET.

Еще раз отметим, что приведенные здесь примеры демонстрируют только самые основные действия. В процессе решения задачи обеспечения взаимодействия с базой данных могут использоваться очень многие варианты и способы оптимизации. В самих примерах кода даны пояснения, касающиеся наиболее важных вопросов, но это всего лишь код, а не подробное описание. Автор настоятельно рекомендует ознакомиться с книгой, посвященной исключительно проблемам обеспечения взаимодействия с базой данных, или прочитать описание этой темы, приведенное в книге автора, которая готовится к выпуску: Professional SQL Server 2005 Programming.

Некоторые общие понятия

Прежде чем приступить к изучению кода, необходимо разобраться в некоторых наиболее важных понятиях. В процессе развития средств взаимодействия с базой данных было создано несколько различных моделей обеспечения связи, в том числе OLEDB, ODBC, а также, безусловно, ADO (некоторые из них уже в основном потеряли свое значение). В наши дни наиболее широко применяется модель обеспечения связи ADO.NET. Но поскольку не исключено появление нововведений в этой области, не следует удивляться тому, что ко времени выхода следующей версии SQL Server будет предложена еще одна, новейшая модель.

Таким образом, различные объектные модели обеспечения связи сменяют друг друга, но некоторые концепции остаются актуальными для каждой объектной модели. Ниже приведены краткие определения основных концепций.

- Соединение. Объект соединения в основном не требует пояснений. Это — объект, который определяет характеристики подключения и непосредственно выполняет подключение к базе данных. Объект соединения содержит информацию о таких параметрах, как имя пользователя, пароль, база данных и сервер, к которому должно быть выполнено подключение. В объекте соединения, как правило, реализованы методы наподобие `connect` и `disconnect`, позволяющие подключиться к базе данных и разорвать соединение.

- Команда. Объект команды содержит информацию о том, какие действия должны быть выполнены в базе данных. Некоторые объектные модели не включают этот объект (или по крайней мере не обнаруживают его явно), но, разумеется, концепция команды всегда должна быть реализована (хотя иногда команда скрывается под видом метода объекта соединения).

- Набор данных. Набор данных содержит результаты выполнения запроса (вернее, содержит их, если запрос возвращает данные). Некоторые выполняемые запросы не возвращают результаты (в качестве примера можно указать простой оператор `INSERT`), но если происходит возврат результатов, то должен быть предусмотрен какой-то набор данных (иногда называемый результирующим набором, или набором записей), в котором представлены полученные результаты. Объекты набора данных, как правило, предоставляют возможность обрабатывать в цикле содержащиеся в них записи (в основном допускается только однонаправленный просмотр, но не исключено, что будет предусмотрена возможность указывать положение искомой записи для обеспечения более надежного позиционирования). Кроме того, объекты набора данных обычно позволяют выполнять операции обновления, вставки и удаления данных.

Примеры, рассматриваемые в настоящем приложении, относятся только к объектной модели обеспечения связи ADO.NET. Тем не менее объекты, описания которых приведены выше, реализованы не только в ADO.NET, но и во всех других объектных моделях, поэтому приведенные примеры также могут оказаться полезными при их изучении.

Применение средств установления соединений, предусмотренных в языке C#

В настоящее время трудно себе представить, как можно обойтись без использования языка C#. Когда я писал свою предыдущую книгу, язык C# фактически находился лишь на этапе внедрения, но, как оказалось, этот язык сумел за короткое время привлечь к себе много сторонников. Это — довольно качественный язык, относительно простой в изучении (и в этом отношении во многом подобный языку VB). Еще одним преимуществом языка C# является то, что в нем реализованы многие принципы, лежащие в основе языка C. Кроме того, в синтаксисе языков C# и C есть много общего (и благодаря этому упрощается переход с одного языка на другой).

Ниже приведен код на языке C#, применяемый для установления соединения.

```
using System;
using System.Data.SqlClient;
class Program
{
   static void Main()
   {
      // Создать некоторые базовые строки, чтобы можно было рассматривать их
      // отдельно от команд, в которых они используются
      string strConnect =
"Data Source=(local);Initial Catalog=master;Integrated Security=SSPI";
      string strCommand =
"SELECT Name, database_id as ID FROM sys.databases";
      SqlDataReader rsMyRS = null;
      SqlConnection cnMyConn = new SqlConnection(strConnect);
      try{
         // Открыть соединение (при этом фактически впервые осуществляется
         // контакт с сервером базы данных)
         cnMyConn.Open();
         // Создать объект команды
         SqlCommand sqlMyCommand = new SqlCommand( strCommand,cnMyConn);
         // Создать результирующий набор
         rsMyRS = sqlMyCommand.ExecuteReader();
         // Вывести полученные данные
         while (rsMyRS.Read())
         {
            // Вывести данные первого столбца (по порядковому номеру).
            // Можно также ссылаться на столбец по имени
            Console.WriteLine (rsMyRS["Name"]);
         }
      }
      finally
      {
         // Выполнить завершающие действия
         if(rsMyRS != null)
         {
            rsMyRS.Close();
         }
```

```
            if(cnMyConn != null)
            {
                cnMyConn.Close();
            }
        }
    }
}
```

Применение средств установления соединений, предусмотренных в языке VB.NET

В данном разделе также приведен довольно упрощенный код, в котором осуществляется импорт нескольких библиотек, открывается соединение, выполняется запрос, а затем все полученные данные выводятся на консоль. Рекомендуем вызвать этот код на выполнение в пошаговом режиме для подробного ознакомления с тем, что происходит на каждом этапе.

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Module Program
    Sub Main()
        ' Создать некоторые базовые строки, чтобы можно было
        ' рассматривать их отдельно от команд, в которых они используются
        Dim strConnect As String = _
         "Data Source=(local);Initial Catalog=master;Integrated Security=SSPI"
        Dim strCommand As String = _
            "SELECT Name, database_id as ID FROM sys.databases"
        Dim rsMyRS As SqlClient.SqlDataReader
        Dim cnMyConn As New SqlClient.SqlConnection(strConnect)
        ' Открыть соединение (при этом фактически впервые осуществляется
        ' контакт с сервером базы данных)
        cnMyConn.Open()
        ' Создать объект команды
        Dim sqlMyCommand As New SqlClient.SqlCommand(strCommand, cnMyConn)
        ' Создать результирующий набор
        rsMyRS = sqlMyCommand.ExecuteReader()
        ' Вывести полученные данные
        Do While rsMyRS.Read
            ' Вывести данные первого столбца (по порядковому номеру).
            ' Можно также ссылаться на столбец по имени
            Console.WriteLine(rsMyRS("Name"))
        Loop
        ' Clean up
        rsMyRS.Close()
        cnMyConn.Close()
    End Sub
End Module
```

Инсталляция и эксплуатация образцовых баз данных

В настоящей книге для иллюстрации рассматриваемых понятий широко используется несколько образцовых баз данных. Безусловно, некоторые из этих баз данных создаются непосредственно в ходе изложения материала, а другие предоставляются в той или другой форме корпорацией Microsoft. В настоящем приложении описано, как должным образом обеспечить инсталляцию требуемых образцов.

Образцовые базы данных, используемые в настоящей книге

Информация об образцовых базах данных, используемых в настоящей книге, приведена в табл. Д.1.

Таблица Д.1. Информация об образцовых базах данных

База данных	Источник	Примечание
`Northwind`	Загружается с Web-узла корпорации Microsoft	Ссылка, с помощью которой можно перейти на страницу загрузки дистрибутива этой базы данных, приведена в документации SQL Server Books Online
`AdventureWorks`	Находится на дистрибутивном компакт-диске SQL Server 2005	По умолчанию не инсталлируется. Эту базу данных обязательно следует инсталлировать перед выполнением упражнений, приведенных в настоящей книге
`Accounting`	Создается в главе 5 и дорабатывается во многих других главах данной книги	Несколько раз модифицируется в ходе изложения материала в книге. Прежде чем приступить к выполнению примеров, в которых используется эта база данных, убедитесь в том, что выполнение примеров, приведенных в предыдущих главах, полностью завершено
`TestDB`	В книге рассматривается целый ряд примеров создания и удаления этой базы данных	Обычно вслед за созданием базы данных `TestDB` вскоре происходит ее удаление. Каждый последующий пример применения этой базы данных является полностью независимым от предыдущих примеров, поэтому обязательно удаляйте предыдущую версию, прежде чем приступить к работе с базой данных `TestDB` в новой главе
`NorthwindTriggers`	Создается с помощью сценария, загружаемого с Web-узла `wrox.com`	Используется для решения некоторых проблем, связанных с триггерами
`NorthwindBulk`	Создается с помощью сценария, загружаемого с Web-узла `wrox.com`	Применяется в качестве базы данных, создаваемой на основе базе данных `Northwind`, но содержит намного больше строк
`Pubs`	Загружается с Web-узла корпорации Microsoft	Еще один весьма небольшой образец базы данных, используемый во многих примерах, которые можно найти в Интернете

Базы данных, предоставляемые корпорацией Microsoft

В поставку SQL Server 2005 входит только один образец базы данных, который можно найти на дистрибутивном компакт-диске, но в настоящей книге широко используется также образцовая база данных, которая была создана гораздо раньше, — `Northwind`.

Многие представители корпорации Microsoft отнеслись без особого восторга к моему выбору, касающемуся использования базы данных `Northwind` в качестве основы для многих примеров, приведенных в настоящей книге. Для меня решение о применении базы данных `Northwind` также оказалось трудным, хотя я не собираюсь распространяться по этому поводу. Но реальность такова, что есть основания считать вновь созданную базу данных `AdventureWorks` мало пригодной для обучения. Несомненно, эта образцовая база данных является более надежной и позволяет составить гораздо более широкий массив примеров по сравнению с применявшимися ранее базами данных. Однако проблема состоит в том, что эта база данных, во-первых, является чрезмерно сложной для того, чтобы на ее основе можно было преподавать начальные понятия баз данных, во-вторых, она еще более далека от

"реального мира", чем другие образцы баз данных, поскольку в ней некоторые средства SQL Server используются гораздо более интенсивно по сравнению с базами данных, действительно применяемыми на практике.

Несомненно, что по прошествии достаточно продолжительного времени в "настоящих" базах данных станут все шире использоваться некоторые из тех весьма впечатляющих средств, которые были впервые введены в версии SQL Server 2005, но я полагаю, что при этом так и не будут достигнуты исключительно широкие масштабы их применения, которые демонстрируются в проекте базы данных `AdventureWorks`. *В связи с этим я решил ограничиться использованием для многих примеров базы данных* `Northwind`, *пусть даже значительно более упрощенной (возможно, иногда чрезмерно упрощенной), и использовать* `AdventureWorks` *только для иллюстрации наиболее усовершенствованных примеров там, где это имеет смысл.*

Ниже приведена информация о том, как подготовить к работе все необходимые образцы баз данных, в том числе давно известную базу данных `Pubs`.

База данных `AdventureWorks`

Важно отметить, что образцовая база данных `AdventureWorks`, хотя и является единственной включенной в состав дистрибутивного компакт-диска, не инсталлируется по умолчанию. Таким образом, во время инсталляции необходимо специально указать, что должна быть установлена эта база данных, так как в противном случае после запуска сервера она будет недоступной (тем не менее для инсталляции образца `AdventureWorks` можно в дальнейшем повторно запустить процедуру инсталляции).

Основная особенность применяемой при этом процедуры инсталляции состоит в том, что должна быть выбрана опция инсталляции "`Custom Install`", а не "`Typical Install`". А после того, как на экране появится окно с приглашением выбрать устанавливаемые компоненты, найдите обозначение базы данных `AdventureWorks` (не спутайте его с обозначением базы данных `AdventureWorksDW`, которая используется в качестве примера хранилища данных), поставьте рядом с этим обозначением отметку, чтобы включить базу данных `AdventureWorks` в состав компонентов, подлежащих установке. В результате указанная база данных станет доступной при первой же попытке обратиться ко вновь установленному серверу.

База данных `Northwind`

База данных `Northwind` представляет собой образец, который использовался корпорацией Microsoft в течение многих лет для комплектования поставляемых ею программных продуктов для баз данных. Фактически эта база данных была уже давно сформирована группой разработчиков СУБД Access, а свой современный вид приобрела в версии SQL Server 7.0. В Web можно найти огромное число примеров, опубликованных разными авторами, в которых используется эта образцовая база данных. Сам автор использует этот образец фактически в каждой главе настоящей книги.

Инсталляция образцовой базы данных `Northwind` немного сложнее по сравнению с базой данных `AdventureWorks`. Прежде всего, необходимо найти ссылку, с помощью которой можно выполнить загрузку дистрибутива базы данных `Northwind`. Для этого автор рекомендует перейти к документации Books Online и выполнить поиск по ключевому слову `Northwind`. Если используется та версия документации Books Online, которая развернута в Web, то полученная в результате поиска ссылка будет всегда указывать на самую современную версию `Northwind`, независимо от того, в

какое место будет в конечном итоге помещена на Web-узле корпорации Microsoft программа инсталляции этого образца.

После загрузки программы инсталляции базы данных Northwind откройте эту программу и вызовите на выполнение.

Программы инсталляции для образцовых баз данных Northwind и Pubs фактически устанавливают только сценарии создания этих образцов (после вызова этих программ инсталляции на выполнение указанные базы данных фактически не будут установлены на сервере). Для создания образцовых баз данных Northwind и Pubs необходимо вручную выполнить действия, кратко описанные ниже.

После завершения работы программы инсталляции найдите набор сценариев в каталоге "SQL Server 2000 Samples" на жестком диске, выбранном для инсталляции (по умолчанию применяется диск C:). В этом каталоге будет находиться большое количество различных файлов, но вы должны дважды щелкнуть на обозначении файла Northwnd.sql, после чего этот файл будет загружен в программе SQL Server Management Console. Как только будет завершена загрузка файла сценария Northwnd.sql в программу Management Console, щелкните на кнопке Execute, в результате чего должна быть автоматически инсталлирована база данных Northwind.

Следует отметить, что по умолчанию файлы данных базы данных Northwind устанавливаются в тот же каталог, в котором установлена база данных master (независимо от того, какое обозначение пути к каталогу было выбрано в качестве используемого по умолчанию для хранения данных во время инсталляции сервера). Необходимо также учитывать то, что для вызова на выполнение сценария Northwnd.sql необходимо иметь права доступа системного администратора (т.е. пользователя sysadmin) к тому серверу, на котором осуществляется инсталляция. Поэтому, если вы собираетесь эксплуатировать базу данных Northwind на сервере производственного назначения или на другом подобном сервере, к которому вы не имеете доступа, то вам придется обратиться к системному администратору, чтобы он установил для вас базу данных Northwind.

База данных Pubs

База данных Pubs используется в настоящей книге не так уж широко. Однако указанная база данных может стать весьма неплохой тестовой площадкой для очень простых запросов. Кроме тех запросов, которые приведены в настоящей книге, некоторые дополнительные примеры применения этой образцовой базы данных можно найти в Web, поэтому никому не помешает всегда иметь эту базу данных в своем распоряжении.

Образцовые базы данных, создаваемые с помощью сценариев

К числу файлов, которые предоставляются для загрузки с сопровождающего Web-узла настоящей книги, входят два сценария создания образцовых баз данных. Необходимо сохранить каждый из этих двух сценариев создания базы данных (NorthwindBulk и NorthwindTriggers) на диске, как и любой другой загружаемый файл, а затем дважды щелкнуть на каждом из них. В результате этого должна быть запущена программа SQL Server Management Console. После этого вызовите соответствующий сценарий на выполнение.

Следует отметить, что особенно большой объем данных, измеряемый количеством строк, вырабатывается при создании базы данных `NorthwindBulk`. *В результате для выполнения сценария создания* `NorthwindBulk` *может потребоваться довольно много времени (в зависимости от быстродействия компьютера, от нескольких минут до одного или двух часов).*

Образцовые базы данных, создаваемые с нуля

Кроме того, по мере изложения материала книги рассматриваются примеры создания нескольких баз данных с нуля.

База данных `Accounting`

База данных `Accounting` впервые создается на основе команд, введенных в главе 5, после чего примеры создания и использования этой базы данных продолжают дорабатываться на протяжении нескольких следующих глав. Для полного ознакомления с этими примерами достаточно просто выполнять один за другим сценарии, приведенные в книге по ходу изложения материала.

Следует учитывать, что база данных `Accounting` *представляет собой объект, создаваемый последовательно. Если какой-то этап создания этого объекта будет пропущен, то в некоторых случаях вы не сможете выполнить пример, приведенный в книге, из-за того, что не получены результаты предшествующего шага. Если вы столкнетесь с такой ситуацией, то единственный выход состоит в том, чтобы вернуться к предыдущим главам и выполнить приведенные в них сценарии, чтобы перевести базу данных в такое состояние, в которой она должна находиться к текущему моменту.*

База данных `TestDB`

База данных `TestDB` является довольно неопределенной. Достаточно сказать, что она создается и уничтожается, поэтому может выглядеть в главе 13 иначе, чем в главе 7. Если при использовании базы данных `TestDB` возникнут проблемы, то, вероятно, ее нужно будет удалить, а затем последовательно обновлять, руководствуясь указаниями, приведенными в той главе, где находится пример, который вы хотите воспроизвести с помощью этой базы данных.

Предметный указатель

Л

М

Н

П

T

У

Ф

Научно-популярное издание

Роберт Виейра

Программирование баз данных Microsoft SQL Server 2005 Базовый курс

Литературный редактор *И.А. Попова*
Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 12.12.2006. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 67,08. Уч.-изд. л. 55,45.
Тираж 3000 экз. Заказ № 3681.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Программистам от программистов

Программирование баз данных Microsoft SQL Server™ 2005

Базовый курс

Эта книга представляет собой исчерпывающее введение в программирование баз данных SQL Server 2005, которое будет полезно как начинающим, так и более опытным пользователям SQL Server. Она была полностью пересмотрена применительно к версии 2005 системы управления базами данных SQL Server и зарекомендовала себя в качестве авторитетного источника справочной и обзорной информации, которым читатель будет пользоваться в течение долгого времени после того, как освоит все необходимое для успешной работы по своей специальности.

Данная книга содержит фундаментальное описание SQL Server 2005, начиная с основных объектов, доступ к которым предоставляется с помощью языка SQL. Каждая следующая глава основана на материале предыдущей, поэтому переход ко все более сложным темам происходит постепенно. Прочитав эту книгу, читатель будет полностью подготовлен к самостоятельной работе с СУБД SQL Server 2005 в качестве программиста и при желании сможет перейти к изучению более сложной литературы, предназначенной для профессионалов.

Некоторые темы, рассматриваемые в книге

- Различные пользовательские функции и триггеры
- Способы создания и модификации таблиц
- Средства управления ключами, написания сценариев и работы с хранимыми процедурами
- Методы программирования, основанные на использовании языка XML
- Принципы применения служб Reporting Services и Integration Services
- Различные вспомогательные средства языка SQL

Для кого предназначена эта книга

Эта книга предназначена для разработчиков всех уровней, работающих с базами данных Microsoft, которым требуется авторитетное руководство по основным синтаксическим определениям, системам и стратегиям, реализованным в версии 2005 системы управления базами данных SQL Server.

Категория	Программирование
Предмет рассмотрения	Разработка программного обеспечения для SQL Server 2005

Издательство "Диалектика"
www.dialektika.com

p2p.wrox.com
Информационный центр для программистов

www.wrox.com

ISBN-13: 978-5-8459-1202-2
ISBN-10: 5-8459-1202-4