Глава 3

Основные сведения о языке T-SQL

Эта глава также главным образом посвящена изложению сведений, составляющих фундамент знаний о СУБД SQL Server. В ней очень кратко рассматриваются наиболее важные операторы языка Transact-SQL (который также сокращенно обозначается как T-SQL). В этой главе, как и во всех первых главах настоящей книги, предполагается, что читатель в значительной степени знаком с рассматриваемой темой, поэтому она предназначена для предоставления обзорного материала, а также для заполнения пробелов в знаниях.

Язык T-SQL — это собственный диалект языка структурированных запросов (Structured Query Language — 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) операционной системы Windows; короче говоря, начиная с этого выпуска T-SQL стал одним из языков .NET. Безусловно, в версии SQL Server 2005 предусмотрена возможность использовать для доступа к базе данных любой язык .NET, но в конечном итоге в основе обращения непосредственно к самим данным всегда лежит какой-то код SQL, поэтому язык T-SQL остается базовым языком при выполнении любых действий в СУБД SQL Server. С каждым выпуском SQL Server многое изменяется, но применительно к теме, рассматриваемой в данной главе, следует отметить, что все в основном осталось неизменным, поскольку наиболее фундаментальные операторы доступа к данным являются практически такими же, как и прежде.

В настоящей главе рассмотрены следующие операторы T-SQL:

- оператор `SELECT`;
- оператор `INSERT`;
- оператор `UPDATE`;
- оператор `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. При этом, как правило, относительное количество выполняемых команд, предназначенных для манипулирования данными (т.е. для чтения и модификации данных), намного превышает количество команд других типов (с помощью которых, например, предоставляются права пользователям или создаются таблицы), поэтому фактически невозможно указать какие-либо более важные средства доступа к базе данных по сравнению с ними.

Кроме того, в языке SQL предусмотрено много операций и ключевых слов, позволяющих уточнять назначение запросов. В настоящей главе рассматриваются операторы, наиболее широко применяемые для доступа к базе данных, включая операторы соединения.

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 основная часть сведений о языке SQL, полученных 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.

Основные сведения об операторе `SELECT`

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)> [[AS] <table alias>]
[[{FULL|INNER|{LEFT|RIGHT} OUTER|CROSS}] JOIN <next table>
[ON <join condition>] [<additional JOIN clause> -]]]
[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])]
[;]
```

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.

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 * действует в основном так же, как и звездочки, применяемые в других программных конструкциях, — он представляет собой символ-заместитель. 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. В большинстве вариантов применения путь доступа INFORMATION_SCHEMA является совместимым со стандартом ANSI (но необходимо учитывать то, что его реализация в СУБД SQL Server имеет свои отличительные особенности). В СУБД SQL Server предусмотрен также набор системных функций, представляющих собой обобщение тех операций, которые могут быть выполнены с помощью такого источника информации, как INFORMATION_SCHEMA. С помощью этих системных функций можно гораздо проще получить необходимую информацию, чем с помощью пути доступа INFORMATION_SCHEMA, но они не совместимы со стандартом ANSI.

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

```
USE AdventureWorks
SELECT LastName FROM Person.Contact;
```

В данном случае оператор `USE AdventureWorks` *представляет собой команду, позволяющую перейти от одной текущей базы данных к другой. Автор ввел этот оператор в приведенный выше сценарий для обеспечения перехода к требуемой базе данных, в которой уже установлена данная конкретная комбинация схемы и таблицы (*`schema.table`*).*

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 19972 строк. В данном случае было решено получить список фамилий всех заказчиков, поэтому и была применена конструкция, предназначенная для выборки всех этих данных.

Этот запрос является довольно несложным, но по мере того как запросы становятся все более длинными, все чаще возникает необходимость предусматривать псевдонимы для тех или иных таблиц; это можно сделать, добавив псевдоним, предназначенный для дальнейшего использования в операторе запроса, после имени таблицы.

```
SELECT LastName FROM Person.Contact c;
```

В рассматриваемом примере псевдоним задан, но фактически с его помощью не осуществляются какие-либо действия. Но в случае применения более сложного оператора этот псевдоним может действительно использоваться для ссылки на ту же таблицу в другой конструкции запроса:

```
SELECT c.LastName FROM Person.Contact c;
```

В этой версии запроса псевдоним `c` был присвоен таблице `Person.Contact`. А в списке выборки, находящемся ближе к началу оператора, было указано, что для выборки данных о фамилиях из таблицы `Person.Contact` должен использоваться этот псевдоним. По-видимому, в таких простых запросах, как этот, применение псевдонимов не имеет смысла, но, как вскоре станет очевидно при рассмотрении примеров соединений, псевдонимы позволяют обеспечить повышение удобства кода для чтения. А в некоторых более сложных вариантах запросов псевдонимы фактически становятся обязательными.

*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 нагрузка, 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. Иными словами, настоятельно рекомендуется предусматривать выборку только тех данных, которые действительно требуются.*

Конструкция JOIN

В предыдущих книгах автора для описания этой конструкции предназначались отдельные главы, поскольку с ее помощью выполняются такие важные операции, как соединения; иными словами, и в этой книге мы должны уделить ей достаточно внимания.

Современные базы данных, как правило, являются нормализованными; в процессе нормализации данные, которые можно было бы хранить и в более крупных таблицах, тем не менее распределяются по многочисленным меньшим таблицам в целях устранения повторяющихся данных, сокращения объема занимаемого дискового пространства, повышения производительности и обеспечения целостности данных. Решение задачи нормализации требует больших усилий и является крайне важным для реляционных баз данных, но после полного достижения целей нормализации обнаруживается также, что необходимые данные приходится извлекать из одной, другой, третьей таблицы и т.д. Соединения полностью предназначены для обеспечения выборки данных из нескольких таблиц и включения этих данных в один результирующий набор. Конструкции JOIN применяются в операторах выборки данных в нескольких разных формах, и от выбора конкретной формы зависит то, как осуществляется взаимодействие таблиц, входящих в соединение. Кроме того, для создания конструкций JOIN применяются два варианта синтаксиса, старый и новый; настоящая глава посвящена в основном описанию нового варианта синтаксиса, а старый вариант рассматривается в самом конце главы (поскольку, прежде чем ознакомиться с ним, необходимо рассмотреть еще несколько понятий).

Основные сведения о 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 в соответствии с условиями, подлежащими проверке по другой таблице. Именно в этих ситуациях невозможно обойтись без применения конструкции JOIN.

Прежде всего необходимо разобраться в том, как именно формируется единственный результирующий набор из информации двух таблиц с помощью конструкции JOIN. Применяемый способ формирования результирующего набора зависит от того, какой из четырех различных видов операторов соединения для этого используется. 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. 3.1).

Таблица 3.1. Одна строка из таблицы Films

FilmID	FilmName	YearMade
1	My Fair Lady	1964

Теперь перейдем к рассмотрению строк из таблицы с данными об актерах, называемой Actors (табл. 3.2).

Таблица 3.2. Две строки из таблицы `Actors`

FilmID	FirstName	LastName
1	Rex	Harrison
1	Audrey	Hepburn

Конструкция `JOIN` позволяет создать одну строку из двух строк, находящихся в полностью отдельных таблицах (табл. 3.3).

Таблица 3.3. Результаты соединения данных из таблиц `Films` и `Actors`

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

Фактически в данном случае из общего количества записей, равного трем, были созданы две записи (одна из исходных записей находится в одной таблице, а две — в другой). Одна и та же запись из таблицы `Films` использовалась несколько раз, по одному разу применительно к данным о каждом актере, к которому она относится (поскольку соединение осуществляется по данным поля `FilmID`).

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.

Внутренние соединения

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. 3.4.

Таблица 3.4. Еще один вариант таблицы `Films`

FilmID	FilmName	YearMade
1	My Fair Lady	1964
2	Unforgiven	1992

А в табл. 3.5 приведен пример таблицы Actors.

Таблица 3.5. Таблица Actors

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. 3.6.

Таблица 3.6. Результаты выполнения операции внутреннего соединения

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.

> Внутренние соединения по своему характеру являются исключительными; под этим подразумевается, что любая строка, для которой нет соответствия в обеих таблицах, неизбежно исключается из окончательного варианта результирующего набора.

Теперь перейдем к рассмотрению самих примеров кода.

Вначале рассмотрим упрощенную версию синтаксиса:

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

Это — синтаксис, предусмотренный стандартом ANSI; он может успешно применяться не только в СУБД SQL Server, но и в СУБД других типов, в то время как старый вариант синтаксиса, о котором уже шла речь в данной главе, такой гарантии не дает (тем не менее он все еще используется многими разработчиками и в наши дни, о чем также еще будет сказано в этой главе).

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

```
SELECT *
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID;
```

Результаты этого запроса занимают слишком много места, чтобы их можно было привести в данной книге, но, вообще говоря, при его выполнении должно быть получено немногим меньше чем 300 строк. В целом в отношении результатов этого запроса необходимо сделать несколько замечаний, приведенных ниже.

- Прежде всего, нам удалось успешно соединить две таблицы, причем фактически выполнено соединение таблицы с самой собой; это позволяет сравнивать различные строки, относящиеся к одной и той же таблице (в данном случае в одном представлении таблица рассматривается как содержащая данные о руководителях, а в другом — как таблица с данными обо всех служащих). Для этого мы обязаны были использовать псевдонимы для обозначения таблицы (в рассматриваемом случае были выбраны псевдонимы `e` и `m`).
- В обеих ссылках на эту таблицу был предусмотрен возврат всех столбцов. А если бы в этих ссылках упоминались разные таблицы, то были бы обнаружены имена столбцов из обеих таблиц. В результатах обнаруживаются данные из обоих наборов столбцов, но отсутствует возможность определить, какие из этих данных относятся к тому или другому (одни и те же имена столбцов встречаются дважды, но в этих столбцах часто представлены разные значения).

Таким образом, в операторах соединения следует более четко давать определения тех результатов, которые должны быть получены с их помощью, и полностью раскрывать логику конструкций `JOIN`:

```
SELECT e.EmployeeID,
    ce.FirstName,
    ce.LastName,
    m.EmployeeID AS ManagerID,
    cm.FirstName AS ManagerFirst,
    cm.LastName AS ManagerLast
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID
INNER JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
INNER JOIN Person.Contact cm
  ON m.ContactID = cm.ContactID;
```

Приведенный выше оператор является довольно сложным, поскольку в нем воплощены почти все концепции, лежащие в основе внутренних соединений, поэтому ниже приведено его подробное описание, и, в частности, каждый из компонентов рассматривается отдельно, начиная со следующей конструкции:

```
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID
```

В этой конструкции воплощен первоначальный вариант соединения, в котором формируется соединение таблицы с самой собой. Таблицу, применительно к которой по каким-то причинам выполняется операция, предусматривающая соединение таблицы самой с собой, принято называть *таблицей, ссылающейся на саму себя*. Поскольку в одном операторе упоминаются два экземпляра одной и той же таблицы, приходится задавать для этой таблицы псевдонимы, чтобы иметь возможность определить, на какой экземпляр данной таблицы мы ссылаемся в той или иной точке запроса.

```
INNER JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
INNER JOIN Person.Contact cm
  ON m.ContactID = cm.ContactID
```

Для получения данных об именах служащих выполняется соединение с таблицей Contact. Характерная особенность проекта базы данных, рассматриваемой в этом примере, состоит в том, что архитектор данных предусмотрел выделение информации о производственных контактах в отдельную таблицу, поскольку не исключено, что служащий компании может также оказаться ее клиентом. Но следует отметить, что соединение с таблицей Contact должно быть выполнено дважды: один раз для формирования соединения с данными об имени служащего и второй раз — для соединения с данными об имени руководителя.

```
SELECT e.EmployeeID,
    ce.FirstName,
    ce.LastName,
    m.EmployeeID AS ManagerID,
    cm.FirstName AS ManagerFirst,
    cm.LastName AS ManagerLast
```

В рассматриваемом операторе заслуживают также внимания некоторые изменения в списке выборки. Прежде всего список выборки был конкретизирован и ограничен только теми столбцами, которые с наибольшей вероятностью должны содержать интересующую нас информацию (в данном случае информацию о том, какие служащие подчиняются тем или другим руководителям). Кроме того, обе таблицы применяются больше чем в одном экземпляре, поэтому необходимо обозначить псевдонимом не просто каждую таблицу, но и каждый экземпляр таблицы, в которой имеются столбцы, предназначенные для формирования выходных данных.

В качестве дополнительной информации следует указать, что по умолчанию применяются внутренние соединения. Кроме того, ключевое слово INNER является необязательным. В действительности на практике может быть обнаружено, что большинство разработчиков программного обеспечения на языке SQL не применяют это ключевое слово, чтобы запрос состоял из меньшего количества слов. Поэтому, учитывая все сказанное выше, можно преобразовать рассматриваемый запрос в следующую форму:

```
SELECT e.EmployeeID,
    ce.FirstName,
    ce.LastName,
    m.EmployeeID AS ManagerID,
    cm.FirstName AS ManagerFirst,
    cm.LastName AS ManagerLast
FROM HumanResources.Employee e
JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID
JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
JOIN Person.Contact cm
  ON m.ContactID = cm.ContactID;
```

В СУБД SQL Server этот запрос обрабатывается столь же успешно и приводит к получению таких же результатов.

Внешние запросы

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, применение внешних запросов часто может привести к повышению производительности при их использовании вместо других способов организации работы, которые могут применяться для получения того же результата.

В соединениях различаются стороны — левая и правая. 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.

Рассмотрим еще раз один из вариантов первой версии запроса, касающегося служащих и руководителей:

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID;
```

Объем формируемых результатов не столь велик, но заслуживает внимания то, какое количество строк возвращает этот запрос; на компьютере автора это значение составило 289 строк (такое значение может быть получено после инсталляции базы данных AdventureWorks в конфигурации, предусмотренной по умолчанию).

После этого внесем в запрос лишь некоторые изменения:

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID
FROM HumanResources.Employee e
LEFT OUTER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID;
```

В данном случае указано, что должны быть включены все строки, относящиеся к левой стороне, LEFT, независимо от того, имеются ли для них соответствия в правой стороне. Применительно к рассматриваемому запросу можно отметить, что в нем сформулировано требование включить все строки с данными о служащих, а также те строки с данными о руководителях, для которых имеется соответствие.

Проверка полученных результатов должна показать, что количество строк увеличилось на одну (в базе данных, которую использовал автор для выполнения рассматриваемого примера, количество строк составило 290). Изучите сами полученные данные и определите, сможете ли вы найти строку, которая не встречалась раньше.

```
EmployeeID  ManagerID
----------- -----------
109         NULL
```

Если вам удастся определить такую строку, то вы обнаружите, что служащий с идентификационным номером 109 никому не подчиняется (по-видимому, это старший руководитель). Обратите особое внимание на то, как в СУБД SQL Server решается задача заполнения столбца данными, поступающими из таблицы, находящейся в правой стороне соединения, если искомые данные отсутствуют — вместо отсутствующих данных подставляется NULL-значение. В дальнейшем будет показано, как проводить поиск NULL-значений в одной из сторон соединения для обнаружения таких ситуаций, как отсутствие строк, согласующихся со строками в другой стороне соединения.

Если данный вариант соединения будет заменен противоположным (путем перехода к применению конструкции `RIGHT JOIN`), то применительно к данному конкретному набору записей будет получен тот же результат, как и в запросе с конструкцией `INNER JOIN`, поскольку в правой стороне соединения (в той стороне, в которой представлены данные о руководителях) нет никаких данных, которых бы не было в левой стороне (где рассматриваются данные о служащих). Но это не означает, что правое соединение действует иначе, чем левое; только в этом случае рассматриваются все строки, находящиеся в правой стороне соединения, а в левой стороне остаются лишь те строки, для которых имеется соответствие.

Полные соединения

Полные соединения можно считать своего рода результатом совместного применения левого и правого соединений. Если применяется соединение с ключевым словом `FULL`, соответствующий оператор равносилен передаваемому СУБД SQL Server указанию, что в результаты должны быть включены все строки с обеих сторон соединения. В базе данных `AdventureWorks` трудно найти какие-либо действительно наглядные примеры, позволяющие продемонстрировать именно этот вариант соединения, но поскольку сами принципы применения полных соединений довольно просты, тем более для тех, кто уже освоил работу с внешними соединениями, подготовим только следующую сравнительно несложную демонстрацию:

```
CREATE TABLE Film
(FilmID            int            PRIMARY KEY,
 FilmName          varchar(20)    NOT NULL,
 YearMade          smallint       NOT NULL
);

CREATE TABLE Actors
(FilmID            int            NOT NULL,
 FirstName         varchar(15)    NOT NULL,
 LastName          varchar(15)    NOT NULL,
  CONSTRAINT PKActors PRIMARY KEY(FilmID, FirstName, LastName)
);

INSERT INTO Film
VALUES
  (1, 'My Fair Lady', 1964);
INSERT INTO Film
```

```
VALUES
  (2, 'Unforgiven', 1992);

INSERT INTO Actors
VALUES
  (1, 'Rex', 'Harrison');
INSERT INTO Actors
VALUES
  (1, 'Audrey', 'Hepburn');
INSERT INTO Actors
VALUES
  (3, 'Anthony', 'Hopkins');
```

Некоторые применяемые выше операторы еще не были описаны в данной книге, но в рассматриваемом примере это не имеет большого значения. Прежде всего, я надеюсь, что многие читатели уже имеют определенный уровень знаний (если же вам требуется освежить в памяти элементарные сведения, то рассмотрите возможность еще раз ознакомиться с книгой *Программирование баз Microsoft SQL Server 2005. Базовый курс*); кроме того, фактически в данном случае производится лишь подготовка образца для работы с изучаемыми в данном разделе полными соединениями, а подробные сведения о некоторых применяемых операторах будут приведены ниже в этой главе и в следующей.

Подготовив необходимые данные, перейдем к выполнению следующего оператора `FULL JOIN` и изучению полученных результатов:

```
SELECT *
FROM Film f
FULL JOIN Actors a
  ON f.FilmID = a.FilmID;
```

Приведенные ниже результаты показывают, что применительно ко всем согласующимся данным выполнено соединение, полностью включены данные, которые присутствуют только в левой стороне (и на месте данных, которые не были обнаружены в правой стороне, приведены NULL-значения), а также включены все данные, присутствующие только в правой стороне (и, безусловно, вместо отсутствующих данных приведены NULL-значения).

```
FilmID      FilmName             YearMade FilmID      FirstName       LastName
----------- -------------------- -------- ----------- --------------- ---------
1           My Fair Lady         1964     1           Audrey          Hepburn
1           My Fair Lady         1964     1           Rex             Harrison
2           Unforgiven           1992     NULL        NULL            NULL
NULL        NULL                 NULL     3           Anthony         Hopkins

(4 row(s) affected)
```

Перекрестные соединения

Последний вариант операции соединения, рассматриваемый в данной главе (перекрестные соединения), существенно отличается от соединений других типов. 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 конструкция `ON` отсутствует.

Итак, предположим, что должен быть проведен мысленный эксперимент, в котором рассматривается возможность участия любого актера в любом кинофильме:

```
SELECT *
FROM Film f
CROSS JOIN Actors a;
```

В результате выполнения этого оператора формируются результаты, в которых каждая строка из таблицы `Film` согласована с каждой строкой с данными об актере из таблицы `Actors`:

```
FilmID      FilmName            YearMade FilmID      FirstName       LastName
----------- ------------------- -------- ----------- --------------- ---------
1           My Fair Lady        1964     1           Audrey          Hepburn
1           My Fair Lady        1964     1           Rex             Harrison
1           My Fair Lady        1964     3           Anthony         Hopkins
2           Unforgiven          1992     1           Audrey          Hepburn
2           Unforgiven          1992     1           Rex             Harrison
2           Unforgiven          1992     3           Anthony         Hopkins

(6 row(s) affected)
```

Теперь необходимо привести пояснения по поводу того, для чего может потребоваться выполнение перекрестных соединений. Разумеется, соединения этого типа не зря включены в состав применяемых программных средств. Но область использования таких операторов является довольно ограниченной. До настоящего времени автору приходилось сталкиваться лишь с тем, что перекрестные соединения применяются в двух описанных ниже ситуациях.

- **Формирование образцов данных.** Перекрестные соединения позволяют подготовить два небольших набора неповторяющихся данных, а затем сформировать сочетания этих двух наборов данных всеми возможными способами для получения гораздо более крупной выборки с уникальными значениями, предназначенной для использования в качестве образца данных.
- **Подготовка данных для научных исследований.** Разумеется, и в этом случае речь идет о подготовке выборок, но такая подготовка выполняется с учетом того, что данные будут использоваться в научных расчетах, для которых исходными данными являются декартовы произведения. Автору приходилось также сталкиваться с тем, что перекрестные соединения использовались в качестве способа подготовки данных для проведения аналитических расчетов некоторых типов. По-видимому, чаще всего подобные аналитические расчеты относятся к области статистики.

Подводя итог, можно отметить, что перекрестные соединения используются довольно редко, но когда в этом возникает необходимость, без них действительно трудно обойтись!

Конструкция `WHERE`

Конструкция `WHERE` позволяет осуществлять выборку данных с учетом заданных условий. До сих пор в данной главе рассматривались только такие операторы, в которых не налагались ограничения, касающиеся получаемой с их помощью информации, поэтому в результате присутствовали данные из всех строк указанной таблицы (таблиц), не считая тех, которые были исключены в силу самого характера оператора соединения. В этом разделе вначале рассматривается выборка данных по условию

с использованием одной таблицы, а затем мы снова вернемся к рассмотрению соединений, но уже в сочетании со вновь введенными конструкциями. 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 пятидесятью штатами.

Следует также учитывать, что операции соединения, в которых не используются критерии выборки, позволяют также получать в полном объеме взаимосвязанные данные.

Теперь мы попытаемся выполнить поиск более конкретной информации. 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 требуются наименование и код товара, а также 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.

Заранее отметим, что в этот раз предусмотрено применение таблицы `Production.Product` (ниже в этой главе приведена информация о том, как узнать состав доступных таблиц, если этой информации еще нет в нашем распоряжении). 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. 3.1, 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.

В этом списке не удается найти столбец `Product Name` (наименование товара), но имеется столбец `Name`, который, по-видимому, нам как раз и требуется (не правда ли, у него 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;
```

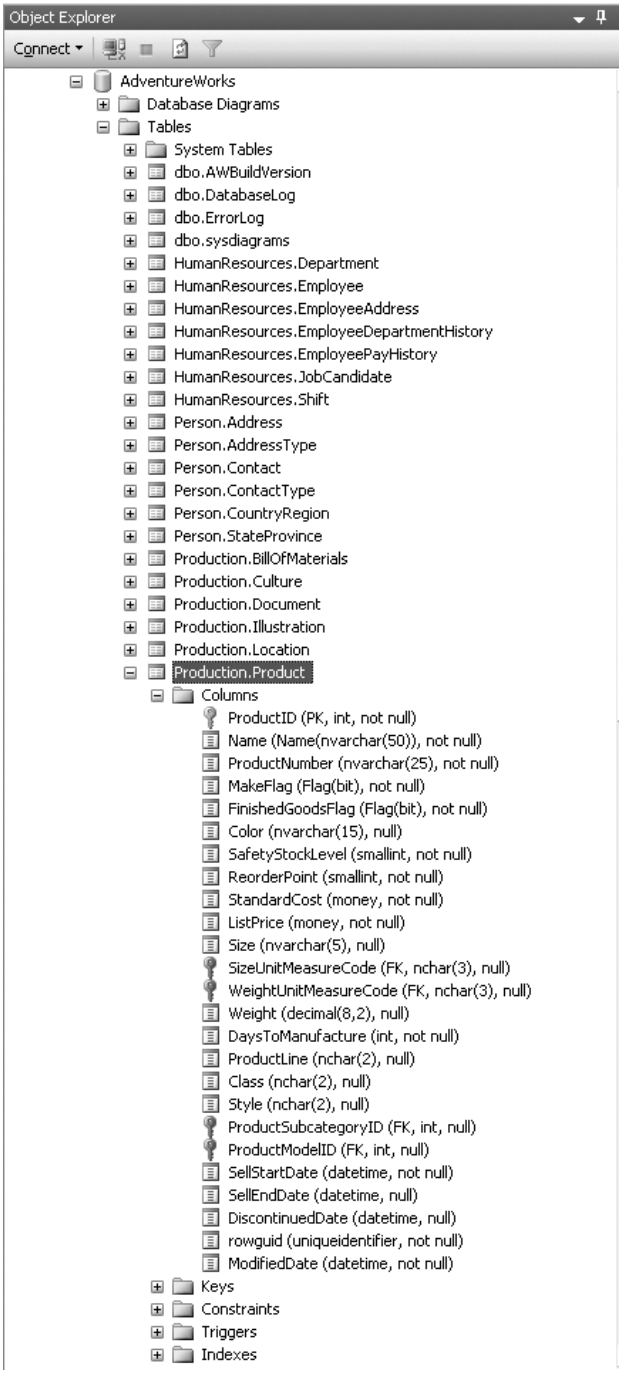

Рис. 3.1. Столбцы таблицы `Production.Product`

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 строки из таблицы, а не только ту, которая необходима.

Такое положение дел вполне бы нас устроило, если бы в таблице было только несколько строк и нам требовалось быстро с ними ознакомиться. Но положение изменяется, если количество строк составляет сотни тысяч или миллионы. Поэтому должна быть предусмотрена условная конструкция, позволяющая включить в полученные результаты запроса данные только об одном товаре с идентификатором 356. Именно для этой цели предназначена конструкция WHERE. Конструкция WHERE следует непосредственно за конструкцией FROM и определяет, каким условиям должна соответствовать строка, для того чтобы ее можно было включить в итоговые результаты. 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;
```

Вызовем на выполнение этот запрос применительно к базе данных AdventureWorks.

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

(1 row(s) affected)
```

Должны были быть получены именно такие результаты. Кроме того, этот запрос был выполнен намного быстрее по сравнению с первым запросом.

Все операции, которые могут использоваться в конструкции WHERE, перечислены в табл. 3.7.

Таблица 3.7. 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 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 <> означают “не равно”. !< и !> означают “не меньше” и “не больше” соответственно

Продолжение табл. 3.7

Операция	Примеры использования	Назначение
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, а после этого 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	Выполнение этой операции сравнения приводит к получению значения TRUE, если первый операнд меньше или равен второму и больше или равен третьему. Это — функциональный эквивалент выражения A>=B AND A<=C. Любое из указанных значений может быть именами столбцов, переменными или литералами
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. Символ _ указывает, что вместо него может быть подставлен любой символ. Конструкция, состоящая из символов, заключенных в квадратные скобки ([]), указывает, что допускается применение любого отдельного символа из числа тех, которые находятся в квадратных скобках (выражение [a-c] показывает, что допускается применение символов a, b и c. Выражение [ab] указывает на возможность применения символа a или 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
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 6, 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

Окончание табл. 3.7

Операция	Примеры использования	Назначение
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 6

Ниже в данном разделе рассматривается тема применения критериев выборки в сочетании с конструкцией JOIN. Фактически для включения в операторы соединения условий выборки не требуется применять каких-либо особых конструкций по сравнению с теми, которые уже были приведены в качестве примеров в рассматриваемых выше операторах с ключевым словом WHERE; достаточно лишь указать необходимые условия в конце оператора. Для уточнения того, к какой таблице относится столбец, применяемый в операторе, можно использовать имя или псевдоним таблицы; это особенно важно, если в разных таблицах имеются столбцы с одинаковыми именами (если имя какого-то столбца является уникальным по отношению ко всем таблицам, то имя столбца может применяться без спецификатора таблицы, причем чаще всего такое условие действительно соблюдается, но автор предпочитает всегда указывать в качестве префикса имя таблицы в условиях конструкции WHERE, хотя бы для того, чтобы знать, из какой таблицы получены данные).

Рассмотрим еще раз тот сложный оператор соединения, который использовался для получения данных обо всех служащих и их руководителях. Но предположим, что на этот раз необходимо включить в список только тех служащих, которыми руководит служащий Jo Brown. Итак, известно, что в качестве условия получения списка применяется то, что в список должны быть включены только подчиненные служащего Jo Brown, поэтому данные о самом этом служащем не должны присутствовать в результирующем наборе; с учетом этого проведено редактирование списка выборки и получен следующий оператор:

```
SELECT e.EmployeeID,
    ce.FirstName,
    ce.LastName
FROM HumanResources.Employee e
JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID
JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
JOIN Person.Contact cm
  ON m.ContactID = cm.ContactID
WHERE cm.FirstName = 'Jo'
  AND cm.LastName = 'Brown';
```

Для решения поставленной задачи нам пришлось применить несколько новых концепций. Прежде всего следует отметить, что ограничения конструкции WHERE распространяются на столбцы, не используемые в списке выборки, а в действительности таблица, упоминаемая в условии, применяется исключительно в качестве источника информации, с помощью которой производится выборка требуемых данных по условию.

Рассмотрим результаты, полученные при выполнении соответствующего запроса:

```
EmployeeID  FirstName                          LastName
----------- ---------------------------------- --------------------------------
1           Guy                                Gilbert
57          Annik                              Stahl
```

```
80          Rebecca                                 Laszlo
89          Margie                                  Shoop
129         Mark                                    McArthur
137         Britta                                  Simon
157         Brandon                                 Heidepriem
162         Jose                                    Lugo
175         Suchitra                                Mohan
213         Chris                                   Okelberry
235         Kim                                     Abercrombie
247         Ed                                      Dudenhoefer

(12 row(s) affected)
```

Итак, результирующий набор, который первоначально состоял из 289 строк, теперь сократился всего лишь до 12.

Конструкция `ORDER BY`

Несложно убедиться в том, что результаты большинства запросов, которые рассматривались до сих пор в настоящей главе, расположены не случайно, а в виде определенной алфавитной или числовой последовательности. Следует отметить, что такое упорядочение результатов складывается не по стечению обстоятельств. Многие начинающие разработчики с удивлением узнают об этом, но действительно так и происходит. Тем не менее, 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`. Этот способ сортировки выбран СУБД SQL Server.

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 задавать 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.

Следует отметить, что операцию сортировки можно было бы также выполнить с использованием числовых полей.

На основании приведенного выше запроса могут быть сделаны неправильные выводы. Изучая приведенное подмножество данных, можно легко стать жертвой заблуждения, что данные отсортированы по полю `ReorderPoint`*, поскольку создается впечатление, что значения данных в этом поле неуклонно уменьшаются, но такая ситуация возникает исключительно в результате случайного совпадения, и анализ всех фактически полученных данных показывает, что значения, определяющие условия возобновления заказа, на самом деле не отсортированы (а в настоящей книге ради сокращения объема изложения приведена лишь часть данных).*

А теперь предположим, что требования к получаемым данным немного изменились, и в связи с этим откорректируем применяемый оператор в целях получения данных еще из одного столбца, `Weight`:

```
SELECT Name, ProductNumber, Weight
FROM Production.Product
WHERE Weight > 800
ORDER BY Weight DESC;
```

После выполнения этого запроса будут получены следующие результаты:

```
Name                               ProductNumber            Weight
---------------------------------- ------------------------ -------------
LL Road Rear Wheel                 RW-R623                  1050.00
ML Road Rear Wheel                 RW-R762                  1000.00
LL Road Front Wheel                FW-R623                  900.00
HL Road Rear Wheel                 RW-R820                  890.00
ML Road Front Wheel                FW-R762                  850.00

(5 row(s) affected)
```

В рассматриваемом запросе заслуживают внимания несколько особенностей. Во-первых, в нем используются многие конструкции, о которых шла речь в этой главе. Мы объединили условие конструкции `WHERE` с конструкцией `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 необходимо получить распечатку всех заказов, которые были размещены в период с 8 по 9 июля 1996 года. Но чтобы усложнить задачу, примем допущение, что заказы требуется отсортировать по дате, а также выполнить еще одну сортировку по значениям столбца `CustomerID`. Кроме того, просто ради интереса примем еще одно небольшое допущение: необходимо в первую очередь расположить заказы, поступившие позже всех (отсортировать по дате в порядке убывания).

Запрос, соответствующий указанным требованиям, должен выглядеть следующим образом:

```
SELECT OrderDate, CustomerID
FROM Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '2001-07-08' AND '2001-07-09'
ORDER BY OrderDate DESC, CustomerID;
```

В данном случае полученные данные отсортированы двумя способами:

```
OrderDate               CustomerID
----------------------- -----------
2001-07-09 00:00:00.000 11025
2001-07-09 00:00:00.000 11238
2001-07-09 00:00:00.000 16629
2001-07-09 00:00:00.000 25861
2001-07-09 00:00:00.000 27577
2001-07-09 00:00:00.000 27666
2001-07-08 00:00:00.000 13258
2001-07-08 00:00:00.000 14560
2001-07-08 00:00:00.000 16607

(9 row(s) affected)
```

Значения CustomerID по-прежнему были отсортированы в порядке возрастания (как принято по умолчанию), поскольку в операторе выборки не содержатся какие-либо иные указания, но, как показывают полученные результаты, заказы, размещенные 9-го числа, расположены перед заказами от 8-го числа, т.е. в порядке убывания даты.

> 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 SalesOrderID, OrderQty
FROM Sales.SalesOrderDetail
WHERE SalesOrderID BETWEEN 43684 AND 43686;
```

Выполнение этого запроса приводит к получению следующего результирующего набора:

```
SalesOrderID OrderQty
------------ --------
43684        2
43684        2
43684        1
43684        2
43684        1
43684        1
43685        3
43685        1
43685        1
43685        1
43686        3
43686        1
43686        1

(13 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 SalesOrderID, SUM(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID BETWEEN 43684 AND 43686
GROUP BY SalesOrderID;
```

Выполнение этого запроса приводит к получению требуемых результатов:

```
SalesOrderID
------------ -----------
43684        9
43685        6
43686        5

(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 ввести запятую и после нее указать имя следующего столбца, в основном по такому же принципу, как и в конструкции ORDER BY.

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. Какие выводы из этого следуют? Рассмотрению этой темы посвящен следующий раздел.

Агрегирующие функции

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 различных агрегирующих функций весьма велико, но в настоящей главе будут рассматриваться только наиболее распространенные.

> 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. К этому описанию трудно что-либо добавить.

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. И в этом случае мы не можем отметить каких-то из ряда вон выходящих особенностей этих функций.

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 HumanResources.Employee
WHERE EmployeeID = 3
```

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 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 HumanResources.Employee
```

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

```
-----------
290

(1 row(s) affected)
```

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

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 HumanResources.JobCandidate;
```

Количество строк в этой таблице меньше, и это отражено в полученных результатах:

```
-----------
13

(1 row(s) affected)
```

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

```
SELECT COUNT(EmployeeID)
FROM HumanResources.JobCandidate;
```

Полученные результаты будут немного отличаться от предыдущих:

```
-----------
2

(1 row(s) affected)

Warning: Null value is eliminated by an aggregate or other SET operation.
```

Чем обусловлено это различие? Причина становится вполне ясной, стоит только изменить свой взгляд на вещи. Дело в том, что не в каждой строке столбца EmployeeID значение присутствует как таковое. Иными словами, функция COUNT при ее использовании в любой форме, отличной от COUNT(*), игнорирует NULL-значения.

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 с NULL-значениями, то эти значения не войдут в состав результатов 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 6, но о нюансах работы с NULL-значениями всегда следует помнить, разрабатывая код T-SQL и проектируя базу данных.

Рассуждения о том, что NULL-значения должны учитываться при проектировании базы данных, не лишены смысла. Достаточно сказать, что решение об использовании NULL-значений в столбце должно приниматься с учетом того, какие запросы будут выполняться по отношению к конкретной базе данных и какие агрегирующие функции будут в них использоваться.

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

Конструкция HAVING используется лишь при том условии, что в запросе присутствует также конструкция GROUP BY; следует учитывать, что конструкция WHERE применяется к каждой строке еще до того, как будет выполнена проверка условий принадлежности этой строки к конкретной группе, а конструкция HAVING применяется к агрегированному значению, относящемуся к этой группе.

Чтобы ознакомиться с тем, в чем это выражается на практике, рассмотрим небольшой пример, используя тот же запрос, который использовался в предыдущем примере конструкции GROUP BY, но добавим конструкцию HAVING:

```
SELECT SalesOrderID, SUM(OrderQty)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID BETWEEN 43684 AND 43686
GROUP BY SalesOrderID
HAVING SUM(OrderQty) > 5;
```

Напомним, что в исходном запросе были возвращены три строки, а на этот раз применение конструкции HAVING приводит к тому, что количество строк сокращается до двух (строка с итоговым значением OrderQty, равным 5, исключена из полученных результатов):

```
SalesOrderID
------------ -----------
```

```
43684        9
43685        6

(2 row(s) affected)
```

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 возможность 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. В настоящей главе подробные сведения об использовании этой конструкции не приведены, поскольку описанию средств языка XML посвящена другая значительная часть данной книги, но следует учитывать, что в главе 16 изложен большой объем материала, касающегося XML. Но во всяком случае опыт автора показывает, 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 главе, в которой речь пойдет о блокировках (глава 12), а поскольку мы еще не описали, как влияют подсказки на выполнение запросов, то не сформированы предпосылки для понимания конструкции OPTION, поэтому отложим обсуждение и этой темы.

Конструкция DISTINCT

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.

Конструкция DISTINCT полностью предназначена для устранения дублирующихся данных. При обнаружении повторяющихся значений любой последующий экземпляр не учитывается (а если конструкция DISTINCT используется вместе с функцией COUNT(), то подсчет уникальных значений также осуществляется единожды). Рассматриваемая конструкция применяется в начале списка выборки (если есть необходимость исключить

из результирующего набора строки с дублирующимися данными) или в операторе COUNT(), если требуется выполнить подсчет количества неповторяющихся значений.

Для ознакомления с особенностями конструкции DISTINCT рассмотрим четыре небольших запроса, которые в основном не требуют пояснений.

Прежде всего выполним выборку из таблицы SalesOrderDetail всех строк со значениями SalesOrderID от 43685 до 43687:

```
SELECT SalesOrderID
FROM Sales.SalesOrderDetail
WHERE SalesOrderID BETWEEN 43685 AND 43687;
```

Выполнение этого оператора приводит к получению следующих результатов:

```
SalesOrderID
------------
43685
43685
43685
43685
43686
43686
43686
43687
43687

(9 row(s) affected)
```

Теперь введем в действие конструкцию DISTINCT:

```
SELECT DISTINCT SalesOrderID
FROM Sales.SalesOrderDetail
WHERE SalesOrderID BETWEEN 43685 AND 43687;
```

В результате ситуация существенно изменяется:

```
SalesOrderID
------------
43685
43686
43687

(3 row(s) affected)
```

Теперь рассмотрим, какое влияние оказывает применение этой конструкции на результаты подсчета количества. Вернемся к первому из рассматриваемых запросов:

```
SELECT COUNT(SalesOrderID)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID BETWEEN 43685 AND 43687;
```

Выполнение этого запроса приводит к получению значения количества, равного девяти:

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

(1 row(s) affected)
```

Но если мы укажем, что в результатах должны учитывать только неповторяющиеся данные:

```
SELECT COUNT(DISTINCT SalesOrderID)
FROM Sales.SalesOrderDetail
WHERE SalesOrderID BETWEEN 43685 AND 43687;
```

```
-----------
3

(1 row(s) affected)
```

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.

Ввод данных с помощью оператора 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.
- За указанными ключевыми словами следует имя таблицы, в которую должны быть вставлены данные.

До сих пор синтаксис оператора вставки не представлял особых сложностей для изучения, а с этого момента мы переходим к рассмотрению немного более трудной темы: 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 находиться 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 этого могут применяться два способа: явно заданные значения и значения, полученные с помощью оператора SELECT.

  Для того чтобы задать 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, явное указание имени каждого столбца способствует уменьшению количества ошибок, а также повышает вероятность того, что разработанные ранее операторы будут оставаться применимыми даже после того, как в дальнейшем в структуру таблицы будут внесены изменения.

Например, предположим, что имеется оператор INSERT, который выглядит примерно так:

```
INSERT INTO HumanResources.JobCandidate
VALUES
  (1, NULL, DEFAULT);
```

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. Исключением из этого правила является ситуация, в которой таблица включает столбец идентификации, поскольку такой столбец при вводе данных не учитывается.

> В частности, ознакомление с определением таблицы HumanRescoures.JobCandidate показывает, что оно начинается с определения столбца идентификации JobCandidateID. Поскольку это — столбец идентификации, известно, что в системе его заполнение осуществляется путем автоматической выработки значения идентификации, поэтому в операторе вставки данных соответствующее значение должно быть пропущено.

Эта книга предназначена для профессиональных разработчиков (а более подробные сведения изложены в книге *Программирование баз Microsoft SQL Server 2005. Базовый курс*), поэтому автор исходит из предположения, что читатель уже знаком с рассмат-

риваемой темой и ему требуется лишь освежить ее в памяти, и в связи с этим ниже приведены лишь краткие замечания.

- Столбец идентификации полностью пропущен (данные в этот столбец вводит система).

Остальные значения заданы явно (в качестве значения `EmployeeID` указано 1, а для столбца `Resume` приведено NULL-значение).

- Применительно к столбцу `ModifiedDate` задано ключевое слово `DEFAULT`, которое служит указанием на то, что в этот столбец должно быть введено значение, предусмотренное по умолчанию.

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

```
INSERT INTO HumanResources.JobCandidate
  (EmployeeID, Resume, ModifiedDate)
VALUES
  (1, NULL, DEFAULT);
```

Еще раз отметим, что значение, относящееся к столбцу идентификации, не задано. Кроме этого, в операторе ввода приведены имена конкретных столбцов; во всем остальном он полностью аналогичен применявшемуся ранее варианту.

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 других данных.
- Та же таблица (в таком случае в операторе `SELECT` обычно предусмотрено выполнение над данными каких-либо математических операций или внесение в данные других изменений).

Оператор `INSERT INTO...SELECT` позволяет решать все эти задачи. 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 выборки данных, хранящихся во временной таблице, который очень часто встречается в практике программирования. 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 10.

```
/* Следующий оператор задает AdventureWorks в качестве текущей базы данных. Это
** позволяет обеспечить переход к использованию нужной базы данных непосредственно
** в коде сценария
*/

USE AdventureWorks;

/* В следующем операторе содержится объявление рабочей таблицы. Данная конкретная
** таблица представляет собой переменную типа таблицы, которая объявляется
** динамически
*/

DECLARE @MyTable Table
(
   SalesOrderID      int,
   CustomerID        int
);

/* 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 SalesOrderID, CustomerID
   FROM AdventureWorks.Sales.SalesOrderHeader
   WHERE SalesOrderID BETWEEN 50222 AND 50225;

-- Наконец, проверим, удалось ли достичь требуемых результатов вставки данных
SELECT *
FROM @MyTable;
```

После выполнения этого кода будут получены следующие результаты:

```
(4 row(s) affected)
SalesOrderID CustomerID
------------ -----------
50222        638
50223        677
50224        247
50225        175

(4 row(s) affected)
```

В данном случае обнаруживаются два сообщения "`(4 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 немного иначе. Дополнительная информация о временных таблицах и переменных типа таблиц приведена в главах 10 и 11.

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. Что под этим подразумевается? В условии выборки и в самом операторе выборки может быть указано несколько разных таблиц (в таком случае для выборки данных применяется соединение), но объектом действия операции обновления может быть одновременно только одна таблица. Рассмотрим несколько простых примеров обновления.

Вначале рассмотрим в качестве примера данные о сотруднике Jo Brown (который уже упоминался в примерах выполнения операции соединения выше в данной главе). В отдел кадров поступила информация о том, что Jo вступил в брак, и теперь необходимо проверить, учтены ли эти сведения в составе данных о сотруднике, хранящихся в базе данных. Ниже приведен запрос, позволяющий получить одну строку данных, содержащую все необходимые данные.

```
SELECT e.EmployeeID,
    e.MaritalStatus,
    ce.FirstName,
    ce.LastName
FROM HumanResources.Employee e
JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
WHERE ce.FirstName = 'Jo'
  AND ce.LastName = 'Brown';
```

Выполнение этого запроса приводит к получению следующих результатов:

```
EmployeeID  MaritalStatus FirstName                LastName
----------- ------------- ---------------------- ----------------------------
16          S             Jo                     Brown

(1 row(s) affected)
```

В столбце MaritalStatus приведено значение "S" (Single — холост), которое уже не соответствует действительности и должно быть заменено более соответствующим истине значением "M" (Married — состоит в браке):

```
UPDATE e
SET MaritalStatus = 'M'
FROM HumanResources.Employee e
JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
WHERE ce.FirstName = 'Jo'
  AND ce.LastName = 'Brown';
```

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:

```
EmployeeID  MaritalStatus FirstName                LastName
----------- ------------- ---------------------- ----------------------------
16          S             Jo                       Brown

(1 row(s) affected)
```

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, для изменения анкетных данных Jo можно было бы применить также следующий оператор:

```
UPDATE e
SET MaritalStatus = 'M', Title = 'Shift Manager'
FROM HumanResources.Employee e
JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
WHERE ce.FirstName = 'Jo'
  AND ce.LastName = 'Brown';
```

При желании в конструкции SET вместо явно заданных значений, которые использовались до сих пор, можно привести выражение. Например, если бы потребовалось увеличить продолжительность отпуска Jo на 20% (учитывая то, каких трудовых успехов добился этот сотрудник), то можно было бы выполнить примерно следующее:

```
UPDATE e
SET VacationHours = VacationHours * 1.2
FROM HumanResources.Employee e
JOIN Person.Contact ce
  ON e.ContactID = ce.ContactID
WHERE ce.FirstName = 'Jo'
  AND ce.LastName = 'Brown';
```

> 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).

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.

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 [TOP (<expression>) [PERCENT]
[FROM ] <table_name>
[FROM ] <table list/JOIN conditions>
[WHERE <search condition>]
```

Наиболее важной отличительной особенностью этого синтаксического определения является то, что в нем конструкция FROM указана дважды (нет, это не опечатка). Первая из этих конструкций может рассматриваться как несколько напоминающая конструкцию FROM оператора обновления, в том смысле, что в ней должно быть указано, применительно к какой таблице должно быть выполнено удаление данных, а вторая в большей степени напоминает конструкцию FROM оператора выборки, поскольку в ней предусмотрены даже условия выполнения соединений (предполагается, что такая форма позволяет проще определить, какие строки должны быть удалены).

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).

Так как первая форма оператора DELETE настолько проста, ниже будет приведен лишь краткий пример ее применения. Для этого воспользуемся данными, которые были введены в базу данных для проведения экспериментов с операторами соединений в начале этой главы. Для удобства ниже еще раз приведен код, который применялся для создания таблиц и ввода в них данных.

Если вы уже подготовили необходимые образцы данных перед выполнением примера с конструкцией FULL JOIN, который рассматривался ранее в этой главе, то можете пропустить этот сценарий создания таблиц и ввода данных, поскольку он идентичен предыдущему.

```
CREATE TABLE Film
(FilmID     int         PRIMARY KEY,
 FilmName   varchar(20) NOT NULL,
 YearMade   smallint    NOT NULL
);

CREATE TABLE Actors
(FilmID         int         NOT NULL,
 FirstName      varchar(15) NOT NULL,
 LastName       varchar(15) NOT NULL,
    CONSTRAINT PKActors PRIMARY KEY(FilmID, FirstName, LastName)
);

INSERT INTO Film
VALUES
```

```
    (1, 'My Fair Lady', 1964);
INSERT INTO Film
VALUES
    (2, 'Unforgiven', 1992);

INSERT INTO Actors
VALUES
    (1, 'Rex', 'Harrison');
INSERT INTO Actors
VALUES
    (1, 'Audrey', 'Hepburn');
INSERT INTO Actors
VALUES
    (3, 'Anthony', 'Hopkins');
```

Прежде всего осуществим выборку данных из таблицы `Film`, исключительно для того, чтобы узнать, какие данные в ней содержатся:

```
SELECT * FROM Film;
```

В результате должны быть получены только те две строки, которые были вставлены в базу данных в сценарии подготовки образца:

```
FilmID      FilmName              YearMade
----------- --------------------- --------
1           My Fair Lady          1964
2           Unforgiven            1992

(2 row(s) affected)
```

Предположим, что теперь необходимо удалить из этой таблицы данные, которые относятся к кинофильму `Unforgiven`.

```
DELETE Film
WHERE FilmName = 'Unforgiven';
```

После этого еще раз выполним тот же оператор `SELECT`:

```
FilmID      FilmName              YearMade
----------- --------------------- --------
1           My Fair Lady          1964

(1 row(s) affected)
```

Мы можем констатировать, что достигнут успех. Та строка, о которой шла речь, действительно была удалена.

Теперь рассмотрим немного более сложный пример, в котором применяется конструкция `JOIN`. Предположим, что на этот раз мы должны удалить из таблицы `Actors` все строки, для которых нет соответствия в таблице `Film`. Для этого необходимо применить запрос, в котором рассматриваются данные из обеих таблиц (иными словами, требуется такой запрос, в котором предусмотрена конструкция `JOIN`). Но, кроме того, необходимо учитывать, что с одной стороны соединения согласование должно отсутствовать (поскольку условием удаления данных является то, что в таблице `Film` отсутствуют данные, согласующиеся с данными о конкретном актере).

Напомним, что для получения NULL-значений в данных, соответствующих той стороне соединения, для которой обнаруживается отсутствие согласования, применяется оператор соединения с конструкцией `OUTER`. Воспользуемся такими результатами для выявления несогласующихся данных, предусмотрев в качестве фактически проверяемого условия проверку на наличие NULL-значений.

Изложение сведений по рассматриваемой теме в этом раздела является довольно кратким, поэтому советуем вам не торопиться и в случае необходимости еще раз возвращаться к прочитанному, особенно если вы впервые сталкиваетесь с этим материалом. Эта книга требует больше знаний и опыта от своих читателей, чем предыдущая книга автора по этой теме, Программирование баз Microsoft SQL Server 2005. Базовый курс*, поскольку изложение материала в ней является чрезвычайно кратким. Еще раз рекомендуем не спешить и тщательно прорабатывать примеры.*

```
DELETE FROM Actors
FROM Actors a
LEFT JOIN Film f
  ON a.FilmID = f.FilmID
WHERE f.FilmID IS NULL;
```

Перейдя сразу же к рассмотрению второй конструкции `FROM`, можно обнаружить, что в ней используется левое соединение. Из этого следует, что будут возвращены данные обо всех актерах. Данные о кинофильмах из таблицы `Film` будут возвращены только в том случае, если имеется согласующийся идентификатор фильма `FilmID`, если же согласование отсутствует, то в тех столбцах, которые содержат данные о кинофильмах, будут присутствовать NULL-значения. В рассматриваемом в данном примере операторе `DELETE` мы исходим из предложенной формулировки условия удаления и проверяем соблюдение этого условия; при обнаружении идентификатора `FilmID`, вместо которого приведено NULL-значение, делается вывод, что согласование в этом конкретном случае отсутствует (поэтому должны быть удалены данные об актере).

Описание альтернативного синтаксиса 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. Несмотря на то, что устаревший синтаксис в наше время все еще довольно широко поддерживается, на всех основных платформах предусмотрена также поддержка синтаксиса ANSI.

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.

Альтернативный вариант конструкции `INNER JOIN`

Вернемся к изложенному выше и еще раз рассмотрим пример применения конструкции `INNER JOIN`, приведенный в этой главе:

```
SELECT *
FROM HumanResources.Employee e
INNER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID;
```

Но вместо использования конструкции `JOIN`, предусмотренной стандартом ANSI, переформулируем этот запрос на основе синтаксиса соединения, предусматривающего применение конструкции `WHERE`. Такая задача действительно является весьма несложной — достаточно исключить слова `INNER JOIN` и ввести запятую, а затем заменить операцию `ON` конструкцией `WHERE`:

```
SELECT *
FROM HumanResources.Employee e, HumanResources.Employee m
WHERE e.ManagerID = m.EmployeeID;
```

На этом внесение изменений заканчивается, а после вызова полученного оператора на выполнение формируются те же строки, 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.).

Альтернативный вариант конструкции OUTER JOIN

С появлением версии SQL Server 2005 возникла такая ситуация, что воспользоваться альтернативным синтаксисом OUTER JOIN можно только при соблюдении определенных условий. Фактически по умолчанию теперь этот синтаксис запрещен, и для его использования необходимо установить уровень совместимости базы данных равным 80 или меньше (в версии SQL Server 2000 он равен 80). К счастью, объем оставшегося кода, в котором по-прежнему используется этот синтаксис, весьма невелик.

Вообще говоря, настоятельно рекомендуется отказаться от этого синтаксиса как полностью устаревшего, поэтому автор не собирается посвящать ему много внимания, но устаревшие конструкции внешнего соединения действуют во много по такому же принципу, как и конструкции внутреннего соединения, если не считать того, что в применявшемся ранее синтаксисе не предусмотрены ключевые слова LEFT или RIGHT (а также, впрочем, OUTER или JOIN), поэтому требуются специальные операции, предусмотренные исключительно для выполнения этой задачи. Указанные знаки операций приведены в табл. 3.8.

Таблица 3.8. Знаки операций и ключевые слова в операторах внешнего соединения

Альтернативный вариант	Вариант 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 e.EmployeeID, m.EmployeeID AS ManagerID
FROM HumanResources.Employee e
LEFT OUTER JOIN HumanResources.Employee m
  ON e.ManagerID = m.EmployeeID;
```

В этом случае также достаточно исключить слова LEFT OUTER JOIN и заменить операцию ON конструкцией WHERE:

```
SELECT e.EmployeeID, m.EmployeeID AS ManagerID
FROM HumanResources.Employee e, HumanResources.Employee m
WHERE e.ManagerID *= m.EmployeeID;
```

> Альтернативный синтаксис для внешних соединений по умолчанию не поддерживается. Для использования этого функционального средства необходимо установить уровень совместимости, меньший или равный 80.

Альтернативный вариант конструкции 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 *
FROM Film f
CROSS JOIN Actors a;
```

Чтобы преобразовать его в оператор с альтернативным синтаксисом, достаточно удалить ключевое слово CROSS JOIN и ввести запятую:

```
SELECT *
FROM Film f, Actors a;
```

Как и при выполнении других примеров данного раздела, будут получены те же результаты, как и с помощью операторов с синтаксисом ANSI.

Итак, теперь описаны конструкции и варианты синтаксиса, применяемые в большинстве систем управления базами данных.

Конструкция 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 (операция объединения), 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. 3.2.

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.

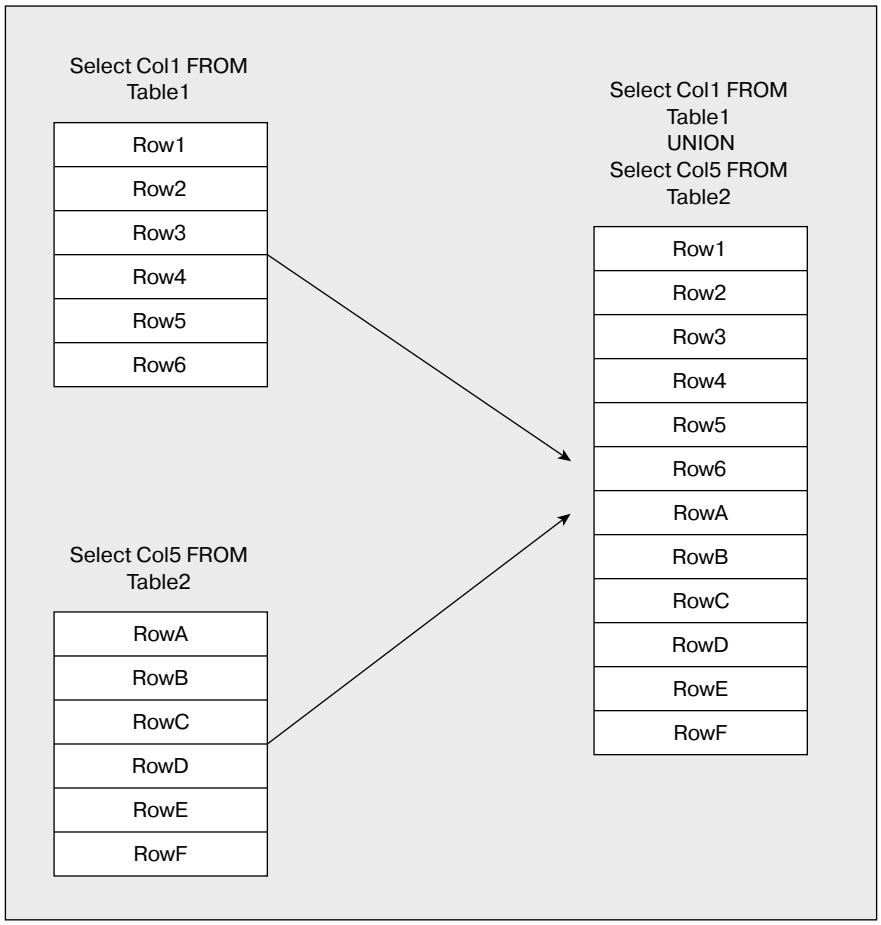

Рис. 3.2. Принцип формирования объединения

- 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 1). 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. Дело в том, что в других запросах происходит возврат всех строк, независимо от того, являются ли они дубликатами по отношению к другой строке или нет, но конструкция UNION действует иначе. Если в запросе с этой конструкцией не используется ключевое слово ALL, то происходит возврат только одной из повторяющихся строк.

И в этом случае проиллюстрируем сказанное на примере.

В данном примере 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');

SELECT *
FROM UnionTest1;

INSERT INTO UnionTest2
VALUES
   ('CCC');
INSERT INTO UnionTest2
VALUES
   ('DDD');
INSERT INTO UnionTest2
VALUES
   ('EEE');

PRINT 'Regular UNION---------------'
SELECT col2
```

```
FROM UnionTest1
  UNION
SELECT col4
FROM UnionTest2;

PRINT 'UNION ALL-------------------'

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).

```
Regular UNION---------------
col2
----
AAA
BBB
CCC
DDD
EEE

(5 row(s) affected)

UNION ALL-------------------
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. Кроме того, удалось добиться того, что с помощью конструкции `UNION` результаты, возвращенные операторами `SELECT`, объединены в один результирующий набор.

Во втором примере наблюдаются лишь незначительные изменения; на этот раз используется конструкция `UNION ALL`, в которой ключевое слово `ALL` обеспечивает получение всех строк. Таким образом, в результатах вновь появляется строка, которая не была возвращена в предыдущем запросе.

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 сменить 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 настоящее время в большинстве программных продуктов с поддержкой SQL по-прежнему допускается применение операторов внутреннего соединения с альтернативным синтаксисом, тогда как операторы внешнего соединения, имеющие альтернативный синтаксис, не поддерживаются. Что же касается соединений с конструкциями `FULL`, то в СУБД, отличных от SQL Server, такие операторы часто вообще не поддерживаются.

В следующей главе приведен общий обзор средств создания и модификации таблиц, применяемых для хранения и выборки данных. Следует отметить, что в этой части средств доступа к базе данных наблюдается гораздо меньше вариантов синтаксиса, не соответствующих стандартным.