SQL Server 2005 T–SQL Recipes: A Problem–Solution Approach

by Joseph Sack

Table of Contents

Chapter 1: Select

Overview

In this chapter, I include recipes for returning data from a SQL Server database using the SELECT statement. At the beginning of each chapter you'll notice that most of the basic concepts are covered first. This is for those of you who are new to the SQL Server 2005 Transact–SQL query language. In addition to the basics, I'll also provide recipes that can be used in your day–to–day development and administration. These recipes will also help you learn about the new functionality introduced in SQL Server 2005. A majority of the examples in this book use the AdventureWorks database, which is an optional install with SQL Server 2005.

Tip The AdventureWorks database is a sample database provided with SQL Server 2005. It's similar to the Northwind and Pubs databases found in previous versions of SQL Server. For instructions on installing this database, see SQL Server 2005 Books Online's topic, "Running Setup to Install AdventureWorks Sample Databases and Samples."

You can read the recipes in this book in almost any order. You can skip to the topics that interest you, or read it through sequentially. If you see something that is useful to you, perhaps a code chunk or example that you can modify for your own purposes or integrate into a stored procedure or function, then this book has been successful.

The Basic SELECT Statement

The SELECT command is the cornerstone of the Transact–SQL language, allowing you to retrieve data from a SQL Server database (and more specifically from database objects within a SQL Server database). Although the full syntax of the SELECT statement is enormous, the basic syntax can be presented in a more boiled down form:

```
SELECT select_list
FROM table_list
```

The select_list argument shown in the previous code listing is the list of columns that you wish to return in the results of the query. The table_list arguments are the actual tables and or views that the data will be retrieved from.

The next few recipes will demonstrate how to use a basic SELECT statement.

Selecting Specific Columns for Every Row

This example demonstrates a very simple SELECT query against the AdventureWorks database, whereby four columns are returned, along with several rows. Explicit column naming is used in the query:

```
USE AdventureWorks
GO

SELECT   ContactID,
      Title,
      FirstName,
      LastName
FROM   Person.Contact
```

The query returns the following abridged results:

```
ContactID   Title    FirstName                                          LastName
----------- -------- -------------------------------------------------- ------------
1           Mr.      Gustavo                                            Achong
2           Ms.      Catherine                                          Abel
3           Ms.      Kim                                                Abercrombie
4           Sr.      Humberto                                           Acevedo
5           Sra.     Pilar                                              Ackerman
...

(19972 row(s) affected)
```

How It Works

The first line of code sets the context database context of the query. Your initial database context, when you first login to SQL Server Management Studio (SSMS), is defined by your login's default database. USE followed by the database name changes your connection context:

```
USE AdventureWorks
GO
```

The SELECT query was used next. The next five lines of code define which four columns to display in the query results:

```
SELECT   ContactID,
      Title,
      FirstName,
      LastName
```

The next line of code is the FROM clause:

```
FROM   Person.Contact
```

The FROM clause is used to specify the data source, which in this example is a table. Notice the two-part name of Person.Contact. The first part (the part before the period) is the *schema* and the second part (after the period) is the actual table name. In SQL Server 2000, the first part of the two part name was called the *object owner*. Now, with SQL Server 2005, users are separated from direct ownership of database objects. Instead of owning the object directly, a *schema* contains the object, and that schema is then *owned* by a user. In SQL Server 2000, if an object was owned by Jane, and Jane left the company, you would not be able to drop her login until you reassigned all of the objects that Jane owned to another user. Now with users owning a schema instead, and the schema containing the object, you can change the owner of the schema and drop Jane's login without having to modify object ownership.

Selecting Every Column for Every Row

If you wish to show *all* columns from the data sources in the FROM clause, you can use the following query:

```
USE AdventureWorks
GO

SELECT   *
FROM   Person.Contact
```

The abridged column and row output is shown here:

```
ContactID   NameStyle   Title   FirstName   MiddleName   LastName
1           0           Mr.     Gustavo     NULL         Achong
2           0           Ms.     Catherine   R.           Abel
3           0           Ms.     Kim         NULL         Abercrombie
...
```

How It Works

The asterisk symbol (*) returns all columns for every row of the table or view you are querying. All other details are as explained in the previous recipe.

Please remember that, as good practice, it is better to explicitly reference the columns you want to retrieve instead of using SELECT *. If you write an application that uses SELECT *, your application may expect the same columns (in the same order) from the query. If later on you add a new column to the underlying table or view, or if you reorder the table columns, you could break the calling application, because the new column in your result-set is unexpected. Using SELECT * can also negatively impact performance, as you may be returning more data than you need over the network, increasing the result set size and data retrieval operations on the SQL Server instance.

Selective Querying Using a Basic WHERE Clause

In a SELECT query, the WHERE clause is used to restrict rows returned in the query result set. The simplified syntax for including the WHERE clause is as follows:

```
SELECT select_list
FROM table_list
[WHERE search_conditions]
```

The WHERE clause uses search conditions which determine the rows returned by the query. Search conditions use predicates, which are expressions that evaluate to TRUE, FALSE, or UNKNOWN. UNKNOWN values can make their appearance when NULL data is used in the search conditions. A NULL value doesn't mean that the value is blank or zero—only that the value is unknown.

The next few recipes will demonstrate how to use the WHERE clause to specify which rows are and aren't returned in the result set.

Using the WHERE Clause to Specify Rows Returned in the Result Set

This basic example demonstrates how to select which rows are returned in the query results:

```
SELECT   Title,
   FirstName,
   LastName
FROM   Person.Contact
WHERE Title = 'Ms.'
```

This example returns the following (abridged) results:

```
Title    FirstName                                          LastName
-------- -------------------------------------------------- --------------------
Ms.      Catherine                                          Abel
Ms.      Kim                                                Abercrombie
Ms.      Frances                                            Adams
Ms.      Margaret                                           Smith
...

(415 row(s) affected)
```

How It Works

In this example, you can see that only rows where the person's title was equal to "Ms." were returned. This search condition was defined in the WHERE clause of the query:

```
WHERE Title = 'Ms.'
```

Only one search condition was used in this case; however, an almost unlimited number of search conditions can be used in a single query, as you'll see in the next recipe.

Combining Search Conditions

This recipe will demonstrate connecting multiple search conditions by utilizing the AND, OR, and NOT logical operators. The AND logical operator joins two or more search conditions and returns the row or rows only when each of the search conditions is true. The OR logical operator joins two or more search conditions and returns the row or rows in the result set when any of the conditions are true.

In this first example, two search conditions are used in the WHERE clause, separated by the AND operator. The AND means that for a given row, both search conditions must be true for that row to be returned in the result set:

```
SELECT   Title,
      FirstName,
      LastName
FROM   Person.Contact
WHERE Title = 'Ms.' AND
      LastName = 'Antrim'
```

This returns the following results:

```
Title    FirstName                                          LastName
-------- -------------------------------------------------- ------------------------
Ms.      Ramona                                             Antrim

(1 row(s) affected)
```

In this second example, an OR operator is used for the two search conditions instead of an AND, meaning that if *either* search condition evaluates to TRUE for a row, that row will be returned:

```
SELECT   Title,
      FirstName,
      LastName
FROM   Person.Contact
WHERE Title = 'Ms.' OR
      LastName = 'Antrim'
```

This returns the following (abridged) results:

```
Title    FirstName                                          LastName
-------- -------------------------------------------------- ------------------------
Ms.      Ramona                                             Antrim
Ms.      Frances                                            Adams
Ms.      Margaret                                           Smith
Ms.      Carla                                              Adams
Ms.      Kim                                                Akers
...
(415 row(s) affected)
```

How It Works

In the first example, two search conditions were joined using the AND operator:

```
WHERE Title = 'Ms.' AND
      LastName = 'Antrim'
```

As you add search conditions to your query, you join them by the logical operators AND and OR. For example, if both the Title equals Ms. *and* the LastName equals Antrim, that row or rows will be returned. The AND operator dictates that *both* joined search conditions must be true in order for the row to be returned.

The OR operator, on the other hand, returns rows if *either* search condition is TRUE, as the third example demonstrated:

```
WHERE Title = 'Ms.' OR
      LastName = 'Antrim'
```

So instead of a single row as the previous query returned, rows with a Title of Ms. or a LastName of Antrim were returned.

Negating a Search Condition

The NOT logical operator, unlike AND and OR, isn't used to combine search conditions, but instead is used to negate the expression that follows it.

This next example demonstrates using the NOT operator for reversing the result of the following search condition and qualifying the Title to be equal to 'Ms.' (reversing it to anything *but* 'Ms.'):

```
SELECT   Title,
      FirstName,
      LastName
FROM   Person.Contact
WHERE NOT Title = 'Ms.'
```

This returns the following (abridged) results:

```
Title    FirstName                                          LastName
-------- -------------------------------------------------- ------------------------
Mr.      Gustavo                                            Achong
Sr.      Humberto                                           Acevedo
Sra.     Pilar                                              Ackerman
...

(594 row(s) affected)
```

How It Works

This example demonstrated the NOT operator:

```
WHERE NOT Title = 'Ms.'
```

NOT specifies the reverse of a search condition, in this case specifying that only rows that *don't* have the Title = 'Ms.' be returned.

Keeping Your WHERE Clause Unambiguous

You can use multiple operators (AND, OR, NOT) in a single WHERE clause, but it is important to keep your intentions clear by properly embedding your ANDs and ORs in parentheses. The AND operator limits the result set, and the OR operator expands the conditions for which rows will be returned. When multiple operators are used in the same WHERE clause, operator precedence is used to determine how the search conditions are evaluated. For example, the NOT operator takes precedence (is evaluated first) before AND. The AND operator takes precedence over the OR operator. Using both AND and OR operators in the same WHERE clause without using parentheses can return unexpected results.

For example, the following query may return unintended results:

```
SELECT   Title,
   FirstName,
   LastName
FROM   Person.Contact
WHERE   Title = 'Ms.' AND
   FirstName = 'Catherine' OR
   LastName = 'Adams'
```

This returns the following (abridged) results:

```
Title    FirstName                                          LastName
-------- -------------------------------------------------- ------------------------
Ms.      Catherine                                          Abel
Ms.      Frances                                            Adams
Ms.      Carla                                              Adams
```

```
Mr.       Jay                                                Adams
...
```

Was the author of this query's intention to return results for all rows with a Title of Ms., and of those rows, only include those with a FirstName of Catherine or a LastName of Adams? Or did the author wish to search for all people named Ms. with a FirstName of Catherine, *as well as* anyone with a LastName of Adams?

A query that uses both AND and OR should use parentheses to clarify exactly what rows should be returned. For example, this next query returns anyone with a Title of Ms. *and* a FirstName equal to Catherine. It also returns anyone else with a LastName of Adams—regardless of Title and FirstName:

```
SELECT   ContactID,
      Title,
      FirstName,
      MiddleName,
      LastName
FROM   Person.Contact
WHERE    (Title = 'Ms.' AND
      FirstName = 'Catherine') OR
      LastName = 'Adams'
```

How It Works

Use parentheses to clarify multiple operator WHERE clauses. Parentheses assist in clarifying a query as they help SQL Server identify the order that expressions should be evaluated. Search conditions enclosed in parentheses are evaluated in an inner to outer order, so in the example from this recipe, the following search conditions were evaluated first:

```
(Title = 'Ms.' AND
FirstName = 'Catherine')
```

before evaluating the outside OR search expression:

```
LastName = 'Adams'
```

Using Operators and Expressions

So far this chapter has used the = (equals) operator to designate what the value of a column in the result set should be. The = comparison operator tests the equality of two expressions. An *expression* is a combination of values, identifiers, and operators evaluated by SQL Server in order to return a result (for example, return TRUE or FALSE or UNKNOWN).

Table 1-1 lists some of the operators you can use in a search condition.

Table 1-1: Operators

Operator	Description
!=	Tests two expressions not being equal to each other.
!>	Tests that the left condition is not greater than the expression to the right.
!<	Tests that the right condition is not greater than the expression to the right.
<	Tests the left condition as less than the right condition.
<=	Tests the left condition as less than or equal to the right condition.
<>	Tests two expressions not being equal to each other.
=	Tests equality between two expressions.
>	Tests the left condition being greater than the expression to the right.

>=	Tests the left condition being greater than or equal to the expression to the right.
ALL	When used with a comparison operator and subquery, if all retrieved values satisfy the search condition, the rows will be retrieved.
ANY	When used with a comparison operator and subquery, if any retrieved values satisfy the search condition, the rows will be retrieved.
BETWEEN	Designates an inclusive range of values. Used with the AND clause between the beginning and ending values.
CONTAINS	Does a fuzzy search for words and phrases.
ESCAPE	Takes the character used prior to a wildcard character to designate that the literal value of the wildcard character should be searched, rather than use the character as a wildcard.
EXISTS	When used with a subquery, EXISTS tests for the existence of rows in the subquery.
FREETEXT	Searches character-based data for words using meaning, rather than literal values.
IN	Provides an inclusive list of values for the search condition.
IS NOT NULL	Evaluates if the value is NOT null.
IS NULL	Evaluates whether the value is null.
LIKE	Tests character string for pattern matching.
NOT BETWEEN	Specifies a range of values NOT to include. Used with the AND clause between the beginning and ending values.
NOT IN	Provides a list of values for which NOT to return rows for.
NOT LIKE	Tests character string, excluding those with pattern matches.
SOME	When used with a comparison operator and subquery, if any retrieved values satisfy the search condition, the rows will be retrieved.

As you can see from Table 1-1, SQL Server 2005 includes several operators which can be used within query expressions. Specifically, in the context of a WHERE clause, operators can be used to compare two expressions, and also check whether a condition is TRUE, FALSE, or UNKNOWN. The next few recipes will demonstrate how the different operators are used within search expressions.

Using BETWEEN for Date Range Searches

This example demonstrates the BETWEEN operator, used to designate sales orders that occurred between the dates 7/28/2002 and 7/29/2002:

```
SELECT   SalesOrderID,
      ShipDate
FROM Sales.SalesOrderHeader
WHERE ShipDate BETWEEN '7/28/2002' AND '7/29/2002'
```

The query returns the following (abridged) results:

```
SalesOrderID ShipDate
------------ -----------------------
46845        2002-07-28 00:00:00.000
46846        2002-07-28 00:00:00.000
46847        2002-07-28 00:00:00.000
more rows
46858        2002-07-29 00:00:00.000
46860        2002-07-29 00:00:00.000
46861        2002-07-29 00:00:00.000

(17 row(s) affected)
```

How It Works

The exercise demonstrated the BETWEEN operator, which tested whether or not a column's ShipDate value fell between two dates:

```
WHERE ShipDate BETWEEN '7/28/2002' AND '7/29/2002'
```

Using Comparisons

This next example demonstrates the < Less Than operator which is used in this query to only show products with a standard cost below $110.00:

```
SELECT   ProductID,
   Name,
   StandardCost
FROM   Production.Product
WHERE    StandardCost < 110.0000
```

This query returns the following (abridged) results:

```
ProductID   Name                                               StandardCost
----------- -------------------------------------------------- ---------------------
1           Adjustable Race                                    0.00
2           Bearing Ball                                       0.00
3           BB Ball Bearing                                    0.00
4           Headset Ball Bearings                              0.00
more rows
952         Chain                                              8.9866
994         LL Bottom Bracket                                  23.9716
995         ML Bottom Bracket                                  44.9506
996         HL Bottom Bracket                                  53.9416
more rows
(317 row(s) affected)
```

How It Works

This example demonstrated the < Less Than operator, returning all rows with a StandardCost less than 110.0000:

```
WHERE    StandardCost < 110.0000
```

Checking for NULL Values

This next query tests for the NULL value of a specific column. A NULL value does *not* mean that the value is blank or zero—only that the value is *unknown*. This query returns any rows where the value of the product's weight is unknown:

```
SELECT   ProductID,
   Name,
   Weight
FROM    Production.Product
WHERE    Weight IS NULL
```

This query returns the following (abridged) results:

```
ProductID   Name                                               Weight
----------- -------------------------------------------------- ---------------------
1           Adjustable Race                                    NULL
2           Bearing Ball                                       NULL
3           BB Ball Bearing                                    NULL
4           Headset Ball Bearings                              NULL
more rows
(299 row(s) affected)
```

How It Works

This example demonstrated the IS NULL operator, returning any rows where the Weight value was unknown (not available):

```
WHERE    Weight IS NULL
```

Returning Rows Based on a List of Values

In this example, the IN operator validates the equality of the Color column to a list of expressions:

```
SELECT   ProductID,
      Name,
      Color
FROM Production.Product
WHERE Color IN ('Silver', 'Black', 'Red')
```

This returns the following (abridged) results:

```
ProductID   Name                                               Color
----------- -------------------------------------------------- -----
317         LL Crankarm                                        Black
318         ML Crankarm                                        Black
319         HL Crankarm                                        Black
more results
988         Mountain-500 Silver, 52                            Silver
790         Road-250 Red, 48                                   Red
```

How It Works

This example demonstrated the IN operator, returning all products that had a Silver, Black, or Red color:

```
WHERE Color IN ('Silver', 'Black', 'Red')
```

Using Wildcards with LIKE

Wildcards are used in search expressions to find pattern matches within strings. In SQL Server 2005, you have the following wildcard options described in Table 1-2.

Table 1-2: Wildcards

Wildcard	Usage
%	Represents a string of zero or more characters.
_	Represents a single character.
[]	Specifies a single character, from a selected range or list.
[^]	Specifies a single character not within the specified range.

This example demonstrates using the LIKE operation with the % wildcard, searching for any product with a name starting with the letter B:

```
SELECT   ProductID,
   Name
FROM   Production.Product
WHERE Name LIKE 'B%'
```

This returns the following results:

```
ProductID   Name
----------- --------------------------------------------------
3           BB Ball Bearing
2           Bearing Ball
```

```
877         Bike Wash - Dissolver
316         Blade
(4 row(s) affected)
```

What if you want to search for the literal value of the % percentage sign or an _ underscore in your character column? For this, you can use the ESCAPE operator.

This next query searches for any product name with a literal _ underscore value in it. The ESCAPE operator allows you to search for the wildcard symbol as an actual character:

```
SELECT ProductID,
      Name
FROM Production.Product
WHERE Name LIKE '%/_%' ESCAPE '/'
```

How It Works

Wildcards allow you to search for patterns in character-based columns. In the example from this recipe, the % percentage sign was used to represent a string of zero or more characters:

```
WHERE Name LIKE 'B%'
```

If searching for a literal value that would otherwise be interpreted by SQL Server as a wildcard, you can use the ESCAPE keyword. The example from this recipe searched for a literal underscore in the Name column:

```
WHERE Name LIKE '%/_%' ESCAPE '/'
```

A backslash embedded in single quotes was put after the ESCAPE command. This designates the backslash symbol as the escape character. If an escape character precedes the underscore within a search condition, it is treated as a literal value instead of a wildcard.

Ordering Results

The ORDER BY clause orders the results of a query based on designated columns or expressions. The basic syntax for ORDER BY is as follows:

```
SELECT select_list
[INTO new_table_name]
FROM table_list
[WHERE search_conditions]
[GROUP BY group_by_list]
[HAVING search_conditions]
[ORDER BY order_list [ASC | DESC] ]
```

ORDER BY must appear after the required FROM clause, as well as the optional WHERE, GROUP BY, and HAVING clauses.

Using the ORDER BY Clause

This example demonstrates ordering the query results by columns ProductID and EndDate:

```
SELECT   p.Name,
      h.EndDate,
      h.ListPrice
FROM   Production.Product p
INNER JOIN Production.ProductListPriceHistory h ON
   p.ProductID = h.ProductID
ORDER BY p.Name, h.EndDate
```

This query returns:

```
Name                                               EndDate                 ListPrice
-------------------------------------------------- ----------------------- ---------
All-Purpose Bike Stand                             NULL                    159.00
AWC Logo Cap                                       NULL                    8.99
AWC Logo Cap                                       2002-06-30 00:00:00.000 8.6442
AWC Logo Cap                                       2003-06-30 00:00:00.000 8.6442
more rows
Women's Tights, L                                  2003-06-30 00:00:00.000 74.99
Women's Tights, M                                  2003-06-30 00:00:00.000 74.99
Women's Tights, S                                  2003-06-30 00:00:00.000 74.99

(395 row(s) affected)
```

The default sorting order of ORDER BY is ascending order, which can be explicitly designated as ASC too. In this next example, DESC is used to return the results in reverse (descending) order:

```
SELECT   p.Name,
      h.EndDate,
      h.ListPrice
FROM   Production.Product p
INNER JOIN Production.ProductListPriceHistory h ON
   p.ProductID = h.ProductID
ORDER BY p.Name DESC, h.EndDate DESC
```

This returns the following abridged results:

```
Name                                               EndDate                 ListPrice
-------------------------------------------------- ----------------------- ---------
Women's Tights, S                                  2003-06-30 00:00:00.000 74.99
Women's Tights, M                                  2003-06-30 00:00:00.000 74.99
Women's Tights, L                                  2003-06-30 00:00:00.000 74.99
...
AWC Logo Cap                                       2002-06-30 00:00:00.000 8.6442
AWC Logo Cap                                       NULL                    8.99
All-Purpose Bike Stand                             NULL                    159.00

(395 row(s) affected)
```

This third example demonstrates ordering results based on a column that is not used in the SELECT clause:

```
SELECT   p.Name
FROM   Production.Product p
ORDER BY p.Color
```

This returns the following abridged results:

```
name
Guide Pulley
LL Grip Tape
ML Grip Tape
HL Grip Tape
Thin-Jam Hex Nut 9
...
```

How It Works

Although queries sometimes appear to return data properly without an ORDER BY clause, the natural ordering of results is determined by the physical key column order in the clustered index (see Chapter 5 for more information on clustered indexes). If the row order of your result sets is critical, you should never depend on the implicit physical order.

In the first example, the Production.Product and Production.ProductListPriceHistory tables were queried to view the history of product prices over time.

Note The full details of INNER JOIN are provided later in the chapter.

The following line of code sorted the results first alphabetically by product name, and then by the end date:

```
ORDER BY p.Name, h.EndDate
```

You can designate one or more columns in your ORDER BY clause, so long as the columns do not exceed 8,060 bytes in total.

The second example demonstrated returning results in descending order (ascending is the default order). The DESC keyword was referenced behind each column that required the descending sort:

```
ORDER BY p.Name DESC, h.EndDate DESC
```

The third example demonstrated ordering the results by a column that was not used in the SELECT statement:

```
ORDER BY p.Color
```

One caveat when ordering by unselected columns is that ORDER BY items must appear in the select list if SELECT DISTINCT is specified.

Using the TOP Keyword with Ordered Results

The TOP keyword allows you to return the first n number of rows from a query based on the number of rows or percentage of rows that you define. The first rows returned are also impacted by how your query is ordered.

Note SQL Server 2005 also provides new ranking functions which can be used to rank each row within the partition of a result set. For a review of ranking functions, see Chapter 8.

In this example, the top ten rows are retrieved from the Purchasing.Vendor table for those rows with the highest value in the CreditRating column:

```
SELECT TOP 10 v.Name,
   v.CreditRating
FROM Purchasing.Vendor v
ORDER BY v.CreditRating DESC, v.Name
```

This returns:

```
Name                                               CreditRating
-------------------------------------------------- ------------
Merit Bikes                                        5
Victory Bikes                                      5
Proseware, Inc.                                    4
Recreation Place                                   4
Consumer Cycles                                    3
Continental Pro Cycles                             3
Federal Sport                                      3
Inner City Bikes                                   3
Northern Bike Travel                               3
Trey Research                                      3

(10 row(s) affected)
```

The next example demonstrates limiting the *percentage* of rows returned in a query using a local variable (using local variables in TOP is a new feature in SQL Server 2005):

```
DECLARE @Percentage float
```

```
SET @Percentage = 1

SELECT TOP (@Percentage) PERCENT
      Name
FROM Production.Product
ORDER BY Name
```

This returns the top 1 percent of rows from the Production.Product table, ordered by product name:

```
Name
--------------------------------------------------
Adjustable Race
All-Purpose Bike Stand
AWC Logo Cap
BB Ball Bearing
Bearing Ball
Bike Wash - Dissolver
(6 row(s) affected)
```

How It Works

In previous versions of SQL Server, developers used SET ROWCOUNT to limit how many rows the query would return or impact. In SQL Server 2005, you should use the TOP keyword instead of SET ROWCOUNT, as the TOP will usually perform faster. Also, *not* having the ability to use local variables in the TOP clause was a major reason why people still used SET ROWCOUNT over TOP in previous versions of SQL Server. With these functionality barriers removed, there is no reason not to start using TOP.

> **Tip** The TOP keyword can also now be used with INSERT, UPDATE, and DELETE statements—something that will not be supported with SET ROWCOUNT future versions of SQL Server. For more information about TOP used in conjunction with data modifications, see Chapter 2.

The key to the first example was the TOP keyword, followed by the number of rows to be returned:

```
SELECT TOP 10 v.Name
```

Also important was the ORDER BY clause, which ordered the results prior to the TOP *n* rows being returned:

```
ORDER BY v.CreditRating DESC, v.Name
```

The second example demonstrated how to use the new local variable assignment functionality with TOP PERCENT:

```
DECLARE @Percentage float

SET @Percentage = 1

SELECT TOP (@Percentage) PERCENT
```

The new local variable functionality allows you to create scripts, functions, or procedures that can determine the number of rows returned by a query based on the value set by the caller, instead of having to hardcode a set TOP number or percentage of rows.

Grouping Data

The GROUP BY clause is used in a SELECT query to determine the groups that rows should be put in. The simplified syntax is as follows:

```
SELECT select_list
FROM table_list
```

```
[WHERE search_conditions]
[GROUP BY group_by_list]
```

GROUP BY follows the optional WHERE clause, and is most often used when aggregate functions are referenced in the SELECT statement (aggregate functions are reviewed in more detail in Chapter 8).

Using the GROUP BY Clause

This example uses the GROUP BY clause to summarize total amount due by order date from the Sales.SalesOrderHeader table:

```
SELECT   OrderDate,
   SUM(TotalDue) TotalDueByOrderDate
FROM   Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '7/1/2001' AND '7/31/2001'
GROUP BY OrderDate
```

This returns the following (abridged) results:

```
OrderDate               TotalDueByOrderDate
----------------------- ---------------------
2001-07-01 00:00:00.000 665262.9599
2001-07-02 00:00:00.000 15394.3298
2001-07-03 00:00:00.000 16588.4572
2001-07-04 00:00:00.000 7907.9768
2001-07-05 00:00:00.000 16588.4572
...
2001-07-27 00:00:00.000 30985.659
2001-07-28 00:00:00.000 21862.8284
2001-07-29 00:00:00.000 19545.3176
2001-07-30 00:00:00.000 15914.584
2001-07-31 00:00:00.000 16588.4572

(31 row(s) affected)
```

How It Works

In this recipe's example, the GROUP BY clause was used in a SELECT query to determine the groups that rows should be put in.

Stepping through the first line of the query, the SELECT clause designated that the OrderDate should be returned, as well as the SUM total of values in the TotalDue column. SUM is an aggregate function. An aggregate function performs a calculation against a set of values (in this case TotalDue), returning a single value (the total of TotalDue by OrderDate):

```
SELECT   OrderDate,
   SUM(TotalDue) TotalDueByOrderDate
```

Notice that a *column alias* for the SUM(TotalDue) aggregation was used. A column alias returns a name for a calculated, aggregated, or regular column. This is another method of sending information to the calling application—allowing you to change the underlying source column without the application being aware. Aside from application situations, providing a separate interface from the table structures can also be useful to other referencing database objects (such as views, functions, and stored procedures, which are demonstrated in future chapters).

In the next part of the query, the Sales.SalesOrderHeader table was referenced in the FROM clause:

```
FROM   Sales.SalesOrderHeader
```

Next, the OrderDate was qualified to return rows for the month of July, and the year 2001:

```
WHERE OrderDate BETWEEN '7/1/2001' AND '7/31/2001'
```

The result set was grouped by OrderDate:

```
GROUP BY OrderDate
```

Note that grouping can occur against one or more columns.

Had the GROUP BY clause been left out of the query, using an aggregate function in the SELECT clause would have raised the following error:

```
Msg 8120, Level 16, State 1, Line 1
Column 'Sales.SalesOrderHeader.OrderDate' is invalid in the select list because
it is not contained in either an aggregate function or the GROUP BY clause.
```

This error is raised because any column that is *not* used in an aggregate function in the SELECT list must be listed in the GROUP BY clause.

Using GROUP BY ALL

By adding the ALL keyword after GROUP BY, all row values are used in the grouping, even if they were not qualified to appear via the WHERE clause.

This example executes the same query as the previous recipe's example, except it includes the ALL clause:

```
SELECT   OrderDate,
   SUM(TotalDue) TotalDueByOrderDate
FROM   Sales.SalesOrderHeader
WHERE OrderDate BETWEEN '7/1/2001' AND '7/31/2001'
GROUP BY ALL OrderDate
```

This returns the following (abridged) results:

```
OrderDate               TotalDueByOrderDate
----------------------- --------------------
2001-07-22 00:00:00.000 42256.626
2004-06-15 00:00:00.000 NULL
2002-01-07 00:00:00.000 NULL
more rows
2002-11-14 00:00:00.000 NULL
2002-08-12 00:00:00.000 NULL
Warning: Null value is eliminated by an aggregate or other SET operation.

(1124 row(s) affected)
```

How It Works

In the results returned by the GROUP BY ALL example, notice that TotalDueByOrderDate was NULL for those order dates not included in the WHERE clause. This does not mean they have zero rows, but instead, that data is not *displayed* for them.

This query also returned a warning along with the results:

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

This means the SUM aggregate encountered NULL values and didn't include them in the total. For the SUM aggregate function, this was okay, however NULL values in other aggregate functions can cause undesired results. For example, the AVG function ignores NULL values but the COUNT function does not. If your query uses both these functions, you may think that the NULL value included in COUNT helps make up the AVG results—which it doesn't.

Selectively Querying Grouped Data Using HAVING

The HAVING clause of the SELECT statement allows you to specify a search condition on a query using GROUP BY and/or an aggregated value. The syntax is as follows:

```
SELECT select_list
[ INTO new_table_name ]
FROM table_list
[ WHERE search_conditions ]
[ GROUP BY group_by_list ]
[ HAVING search_conditions ]
```

The HAVING clause is used after the GROUP BY clause. The WHERE clause, in contrast, is used to qualify the rows that are returned *before* the data is aggregated or grouped. HAVING qualifies the aggregated data *after* the data has been grouped or aggregated.

This example queries two tables, Production.ScrapReason and Production.WorkOrder. The Production.ScrapReason is a lookup table that contains manufacturing failure reasons, while the Production.WorkOrder table contains the manufacturing work orders that control which products are manufactured in the quantity and time period, in order to meet inventory and sales needs.

This example reports to management which “failure reasons” have occurred fifty or more times:

```
SELECT  s.Name,
   COUNT(w.WorkOrderID) Cnt
FROM Production.ScrapReason s
INNER JOIN Production.WorkOrder w ON
   s.ScrapReasonID = w.ScrapReasonID
GROUP BY s.Name
HAVING COUNT(*)>50
```

This query returns:

```
Name                                               Cnt
-------------------------------------------------- -----------
Gouge in metal                                     54
Stress test failed                                 52
Thermoform temperature too low                     63
Trim length too long                               52
Wheel misaligned                                   51

(5 row(s) affected)
```

How It Works

In this recipe, the SELECT clause requested a count of WorkOrderIDs by failure name:

```
SELECT  s.Name,
      COUNT(w.WorkOrderID)
```

Two tables were joined by the ScrapReasonID column:

```
FROM Production.ScrapReason s
INNER JOIN Production.WorkOrder w ON
   s.ScrapReasonID = w.ScrapReasonID
```

Since an aggregate function was used in the SELECT clause, the non-aggregated columns must appear in the GROUP BY Clause:

```
GROUP BY s.Name
```

Lastly, using the HAVING query determines that, *of the selected and grouped data*, only those rows in the result set with a count of fifty or higher will be returned:

```
HAVING COUNT(*)>50
```

SELECT Clause Techniques

The SELECT clause is primarily used to define which columns are returned in the result set, but its functionality isn't limited to just that. This next set of queries will detail a number of SELECT clause techniques, including:

- Using the DISTINCT keyword to remove duplicate values
- Renaming columns using column aliases
- Concatenating string values into a single column
- Creating a SELECT statement that itself creates an executable Transact-SQL script
- Creating a comma-delimited array list of values

Using DISTINCT to Remove Duplicate Values

The default behavior of a SELECT statement is to use the ALL keyword (although because it is the default, you'll rarely see this being used in a query), meaning that duplicate rows will be retrieved and displayed if they exist. Using the DISTINCT keyword instead of ALL allows you to only return unique rows in your results.

This example shows you how to use the DISTINCT keyword to remove duplicate values from a set of selected columns, so that only unique rows appear:

```
SELECT   DISTINCT HireDate
FROM   HumanResources.Employee
```

The results show all unique hire dates from the HumanResources.Employee table:

```
HireDate
-----------------------
1996-07-31 00:00:00.000
1997-02-26 00:00:00.000
1997-12-12 00:00:00.000
1998-01-05 00:00:00.000
more rows
2002-11-01 00:00:00.000
2003-04-15 00:00:00.000
2003-07-01 00:00:00.000

(164 row(s) affected)
```

How It Works

Use the DISTINCT keyword to return distinct values in the result set. In this recipe, DISTINCT was used to return unique HireDate column values. Be sure to only use DISTINCT when actually needed or necessary, as it can slow the query down on larger result sets.

Using DISTINCT in Aggregate Functions

You can also use DISTINCT for a column that is used within an aggregate function (aggregate functions are reviewed in more detail in Chapter 8). You may wish to do this in order to perform aggregations on only the unique values of a column.

For example, if you wanted to calculate the average product list price, you could use the following query:

```
SELECT   AVG(ListPrice)
FROM Production.Product
```

This returns:

```
---------------------
438.6662

(1 row(s) affected)
```

But the previous query calculated the average list price across *all* products. What if some product types are more numerous than others? What if you are only interested in the average price of *unique* price points?

In this case you would write the query as:

```
SELECT   AVG(DISTINCT ListPrice)
FROM Production.Product
```

This returns the unique set of price points first, *and then* averages them:

```
---------------------
437.4042

(1 row(s) affected)
```

How It Works

DISTINCT can be used to return unique rows from a result set, as well as force unique column values within an aggregate function. In this example, the DISTINCT keyword was put within the parentheses of the aggregate function.

Using Column Aliases

For column computations or aggregate functions, you can use a column alias to explicitly name the columns of your query output. You can also use column aliases to rename columns that already *have* a name, which helps obscure the underlying column from the calling application (allowing you to swap out underlying columns without changing the derived column name). You can designate a column alias by using the AS keyword, or by simply following the column or expression with the column alias name.

This example demonstrates producing column aliases using two different techniques:

```
SELECT   Color AS 'Grouped Color',
      AVG(DISTINCT ListPrice) AS 'Average Distinct List Price',
      AVG(ListPrice) 'Average List Price'
FROM Production.Product
GROUP BY Color
```

This returns the following abridged results:

```
Grouped Color   Average Distinct List Price Average List Price
--------------- --------------------------- ---------------------
Silver          726.2907                    850.3053
Grey            125.00                      125.00
...
Silver/Black    61.19                       64.0185

(10 row(s) affected)
```

How It Works

This example shows three examples of using column aliasing. The first example demonstrated how to rename an *existing* column using the AS clause. The AS clause is used to change a column

name in the results, or add a name to a derived (calculated or aggregated) column:

```
SELECT Color AS 'Grouped Color',
```

The second example demonstrated how to add a column name to an aggregate function:

```
AVG(DISTINCT ListPrice) AS 'Average Distinct List Price',
```

The third example demonstrated how to add a column alias without using the AS keyword (it can simply be omitted):

```
AVG(ListPrice) 'Average List Price'
```

Using SELECT to Create a Script

As a DBA or developer, you sometimes need a Transact-SQL script to run against several objects within a database or against several databases across a SQL Server instance. Or perhaps you have a very large table with several columns, which you need to validate in search conditions, but you don't want to have to hand type each column.

This next recipe offers a time-saving technique, using SELECT to write out Transact-SQL for you. You can adapt this recipe to all sorts of purposes.

In this example, assume that you wish to check for rows in a table where all values are NULL. There are many columns in the table, and you want to avoid hand-coding them. Instead, you can create a script to do the work for you:

```
SELECT column_name + ' IS NULL AND '
FROM INFORMATION_SCHEMA.columns
WHERE table_name = 'Employee'
ORDER BY ORDINAL_POSITION
```

This returns code that you can integrate into a WHERE clause (after you remove the trailing AND at the last WHERE condition):

```
------------------------------
EmployeeID IS NULL AND
NationalIDNumber IS NULL AND
ContactID IS NULL AND
LoginID IS NULL AND
ManagerID IS NULL AND
Title IS NULL AND
BirthDate IS NULL AND
MaritalStatus IS NULL AND
Gender IS NULL AND
HireDate IS NULL AND
SalariedFlag IS NULL AND
VacationHours IS NULL AND
SickLeaveHours IS NULL AND
CurrentFlag IS NULL AND
rowguid IS NULL AND
ModifiedDate IS NULL AND

(16 row(s) affected)
```

How It Works

The example used string concatenation and the INFORMATION_SCHEMA.columns system view to generate a list of columns from the Employee table. For each column, IS NULL AND was concatenated to its name. The results can then be copied to the WHERE clause of a query, allowing you to query for rows where each column has a NULL value.

This general technique of concatenating SQL commands to various system data columns can be used in numerous ways, including for creating scripts against tables or other database objects. Do be careful when scripting an action against multiple objects or databases—make sure that the

change is what you intended, and that you are fully aware of the script's outcome.

Performing String Concatenation

String concatenation is performed by using the + operator to join two expressions, as this example demonstrates:

```
SELECT   'The ' +
      p.name +
      ' is only ' +
      CONVERT(varchar(25),p.ListPrice) +
      '!'
FROM   Production.Product p
WHERE   p.ListPrice between 100 AND 120
ORDER BY p.ListPrice
```

This returns:

```
-------------------------------------------------------------------------------
The ML Bottom Bracket is only 101.24!
The ML Headset is only 102.29!
The Rear Brakes is only 106.50!
The Front Brakes is only 106.50!
The LL Road Rear Wheel is only 112.57!
The Hitch Rack - 4-Bike is only 120.00!

(6 row(s) affected)
```

How It Works

When used with character data types, the + operator is used to concatenate expressions together. In this example, literal values were concatenated to columns from the Production.Product table. Each row formed a sentence celebrating the low price of each row's product.

String concatenation is often used when generating end-user reports (such as displaying the First and Last Name in a single column), or when you need to combine multiple data columns into a single column (as you'll see in the next recipe).

Creating a Comma Delimited List Using SELECT

This next recipe demonstrates how to create a comma delimited list using a SELECT query. You can use this recipe in several ways. For example, you could integrate it into a user-defined function that returns a comma delimited list of the regions that a salesperson sells to into a single column (see Chapter 11).

This example demonstrates returning one-to-many table data into a single presentable string:

```
DECLARE @Shifts varchar(20)
SET @Shifts = ''

SELECT   @Shifts = @Shifts + s.Name + ','
FROM   HumanResources.Shift s
ORDER BY s.EndTime

SELECT @Shifts
```

This query returns:

```
--------------------
Night,Day,Evening,

(1 row(s) affected)
```

How It Works

In the first part of this script, a local variable was created to hold a comma delimited list:

```
DECLARE @Shifts varchar(20)
```

After a variable is declared, but before it is set, it is given a NULL value. Because we cannot concatenate NULL values with strings, the variable should be set to an initial blank value instead, as was done in the recipe:

```
SET @Shifts = ''
```

In the query itself, a list of shifts are gathered from the HumanResources.Shift table, ordered by EndTime. At the core of this example, you see that the local variable is assigned to the value of itself concatenated to the shift name, and then concatenated to a comma. The query loops through each value ordered by EndTime, appending each one to the local variable:

```
SELECT   @Shifts = @Shifts + s.Name + ','
FROM   HumanResources.Shift s
ORDER BY s.EndTime
```

SELECT is used to display the final contents of the local variable:

```
SELECT @Shifts
```

Using the INTO Clause

The INTO clause of the SELECT statement allows you to create a new table based on the columns and rows of the query results. Ideally you should be creating your tables using the CREATE TABLE command: however using INTO provides a quick-and-dirty method of creating a new table without having to explicitly define the column names and data types.

The INTO clause allows you to create a table in a SELECT statement based on the columns and rows the query returns. The syntax for INTO is as follows:

```
SELECT select_list
[INTO new_table_name]
FROM table_list
```

The INTO clause comes after the SELECT clause but before the FROM clause, as the next recipe will demonstrate.

In this first example, a new table is created based on the results of a query:

```
SELECT   CustomerID,
   Name,
   SalesPersonID,
   Demographics
INTO   Store_Archive
FROM   Sales.Store
```

The query returns the number of rows inserted into the new Store_Archive table, but does not return query results:

```
(701 row(s) affected)
```

In the second example, a table is created without inserting rows into it:

```
SELECT   CustomerID,
      Name,
      SalesPersonID,
      Demographics
INTO   Store_Archive
FROM   Sales.Store
WHERE   1=0
```

This returns the number of rows inserted into your new Store_Archive table (which in this case is zero):

```
(0 row(s) affected)
```

How It Works

This recipe's example looked like a regular SELECT query, only between the SELECT and FROM clauses the following instructions were inserted:

```
INTO   Store_Archive
```

The INTO clause is followed by the new table name (which must not already exist). This can be a permanent, temporary, or global temporary table (See Chapter 4 for more information). The columns you select determine the structure of the table.

This is a great technique for quickly "copying" the base table structure and data of an existing table. Using INTO, you are not required to pre-define the new table's structure explicitly (for example, you do not need to issue a CREATE TABLE statement).

Caution Although the structure of the selected columns is reproduced, the constraints, indexes, and other separate objects dependent on the source table are *not* copied.

In the second example, a new table was created without also populating it with rows. This was achieved by using a WHERE clause condition that always evaluates to FALSE:

```
WHERE   1=0
```

Since the number 1 will never equal the number 0, no rows will evaluate to TRUE, and therefore no rows will be inserted into the new table. However, the new table is created anyway.

SubQueries

A subquery is a SELECT query that is nested within another SELECT, INSERT, UPDATE, or DELETE statement. A subquery can also be nested inside another subquery. Subqueries can often be re-written into regular JOINs, however sometimes an existence subquery (demonstrated in this recipe) can perform better than equivalent non-subquery methods.

A *correlated* subquery is a subquery whose results depend on the values of the outer query.

Using Subqueries to Check for the Existence of Matches

This first example demonstrates checking for the existence of matching rows within a correlated subquery:

```
SELECT DISTINCT s.PurchaseOrderNumber
FROM Sales.SalesOrderHeader s
WHERE EXISTS (   SELECT SalesOrderID
         FROM Sales.SalesOrderDetail
         WHERE UnitPrice BETWEEN 1000 AND 2000 AND
               SalesOrderID = s.SalesOrderID)
```

This returns the following abridged results:

```
PurchaseOrderNumber
-------------------------
PO8410140860
PO12325137381
PO1160166903
PO1073122178
```

```
...
PO15486173227
PO14268145224

(1989 row(s) affected)
```

This second example demonstrates a regular non-correlated subquery:

```
SELECT   SalesPersonID,
      SalesQuota CurrentSalesQuota
FROM      Sales.SalesPerson
WHERE      SalesQuota IN
            (SELECT MAX(SalesQuota)
             FROM Sales.SalesPerson)
```

This returns the three salespeople who had the maximum sales quota of 300,000:

```
SalesPersonID CurrentSalesQuota
------------- ---------------------
275           300000.00
279           300000.00
287           300000.00
Warning: Null value is eliminated by an aggregate or other SET operation.

(3 row(s) affected)
```

How It Works

The critical piece of the first example was the subquery in the WHERE clause, which checked for the existence of SalesOrderIDs that had products with a UnitPrice between 1000 and 2000. A JOIN was used in the WHERE clause of the subquery, between the outer query and the inner query, by stating SalesOrderID = s.SalesOrderID. The subquery used the SalesOrderID from each returned row in the outer query.

In the second example, there is no WHERE clause in the subquery used to join to the outer table. It is not a correlated subquery. Instead, a value is retrieved from the query to evaluate against in the IN operator of the WHERE clause.

Querying from More Than One Data Source

The previous recipes retrieved data from a single table. Most normalized databases have more than one table in them, so more often than not you'll need to retrieve data from multiple tables using a single query. The JOIN keyword allows you to combine data from multiple tables and/or views into a single result set. It joins a column or columns from one table to another table, evaluating whether there is a match.

With the JOIN keyword, you join two tables based on a join condition. Most often you'll see a join condition testing the equality of one column in one table compared to another column in the second table (joined columns do not need to have the same name, only compatible data types).

Tip As a query performance best practice, try to avoid having to convert data types of the columns in your join clause (using CONVERT or CAST, for example). Opt instead for modifying the underlying schema to match data types (or convert the data beforehand in a separate table, temp table, table variable, or Common Table Expression (CTE)).

SQL Server 2005 join types fall into three categories: *inner*, *outer*, and *cross.* Inner joins use the INNER JOIN keywords. INNER JOIN operates by matching common values between two tables. Only table rows satisfying the join conditions are used to construct the result set. INNER JOINs are the default JOIN type, so if you wish, you can use just the JOIN keyword in your INNER JOIN operations.

Outer joins have three different join types: LEFT OUTER, RIGHT OUTER, and FULL OUTER joins. LEFT OUTER and RIGHT OUTER JOINs, like INNER JOINs, return rows that match the conditions of the join condition. *Unlike* INNER JOINs, LEFT OUTER JOINs return unmatched rows from the first table of the join pair, and RIGHT OUTER JOINs return unmatched rows from the second table of the join pair. The FULL OUTER JOIN clause returns unmatched rows on both the left *and* right tables.

A lesser used join type is CROSS JOIN. A CROSS JOIN returns a Cartesian product when a WHERE clause isn't used. A *Cartesian product* produces a result set based on every possible combination of rows from the left table, multiplied against the rows in the right table. For example, if the Stores table has 7 rows, and the Sales table has 22 rows, you would receive 154 rows (or 7 times 22) in the query results (each possible combination of row displayed).

The next few recipes will demonstrate the different join types.

Using INNER Joins

This inner join joins three tables in order to return discount information on a specific product:

```
SELECT   p.Name,
      s.DiscountPct
FROM Sales.SpecialOffer s
INNER JOIN Sales.SpecialOfferProduct o ON
   s.SpecialOfferID = o.SpecialOfferID
INNER JOIN Production.Product p ON
   o.ProductID = p.ProductID
WHERE p.Name = 'All-Purpose Bike Stand'
```

The results of this query:

```
Name                                               DiscountPct
-------------------------------------------------- ---------------------
All-Purpose Bike Stand                             0.00

(1 row(s) affected)
```

How It Works

A join starts after the first table in the FROM clause. In this example, three tables were joined together: Sales.SpecialOffer, Sales.SpecialOfferProduct, and Production.Product.

Sales.SpecialOffer, the first table referenced in the FROM clause, contains a lookup of sales discounts:

```
FROM Sales.SpecialOffer s
```

Notice the letter "s" which trails the table name. This is a table alias. Once you begin using more than one table in a query, it is important to understand the data source of the individual columns. If the same column names exist in two different tables, you could get an error from the SQL compiler asking you to clarify which column you really wanted to return.

As a best practice, it is a good idea to use aliases whenever column names are specified in a query. For each of the referenced tables, a character was used to symbolize the table name—saving you the trouble of spelling it out each time. This query used a single character as a table alias, but you can use any valid identifier. A *table alias*, aside from allowing you to shorten or clarify the original table name, allows you to swap out the base table name if you ever have to replace it with a different table or view, or if you need to self-join the tables. Table aliases are optional, but recommended when your query has more than one table. A table alias follows the table name in the statement FROM clause.

But back to the example… The INNER JOIN keywords followed the first table reference, then the table being joined to it, followed by its alias:

```
INNER JOIN Sales.SpecialOfferProduct o
```

After that, the ON keyword prefaces the column joins:

```
ON
```

This particular INNER JOIN is based on the equality of two columns—one from the first table and another from the second:

```
s.SpecialOfferID = o.SpecialOfferID
```

Next, the Production.Product table is INNER JOIN'd too:

```
INNER JOIN Production.Product p ON
   o.ProductID = p.ProductID
```

Lastly, a WHERE clause is used to filter rows returned:

```
WHERE Name = 'All-Purpose Bike Stand'
```

Using OUTER Joins

This recipe compares the results of an INNER JOIN versus a LEFT OUTER JOIN. This first query displays the tax rates states and provinces using the Person.StateProvince table and the Sales.SalesTaxRate table. The following query uses an INNER JOIN:

```
SELECT   s.CountryRegionCode,
      s.StateProvinceCode,
      t.TaxType,
      t.TaxRate
FROM Person.StateProvince s
INNER JOIN Sales.SalesTaxRate t ON
   s.StateProvinceID = t.StateProvinceID
```

This returns the following (abridged) results:

```
CountryRegionCode StateProvinceCode TaxType TaxRate
----------------- ----------------- ------- ---------------------
CA                AB                1       14.00
CA                ON                1       14.25
CA                QC                1       14.25
CA                AB                2       7.00
more rows
```

But with the INNER JOIN, you are only seeing those records from Person.StateProvince that *have* rows in the Sales.SalesTaxRate table. In order to see *all* rows from Person.StateProvince, whether or not they have associated tax rates, LEFT OUTER JOIN is used:

```
SELECT   s.CountryRegionCode,
      s.StateProvinceCode,
      t.TaxType,
      t.TaxRate
FROM Person.StateProvince s
LEFT OUTER JOIN Sales.SalesTaxRate t ON
   s.StateProvinceID = t.StateProvinceID
```

This returns the following (abridged) results:

```
CountryRegionCode StateProvinceCode TaxType TaxRate
----------------- ----------------- ------- ---------------------
CA                AB                1       14.00
CA                AB                2       7.00
US                AK                NULL    NULL
US                AL                NULL    NULL
more rows
```

How It Works

This recipe's example demonstrated an INNER JOIN query versus a LEFT OUTER JOIN query. The LEFT OUTER JOIN query returned unmatched rows from the first table of the join pair. Notice how this query returned NULL values for those rows from Person.StateProvince that didn't have associated rows in the Sales.SalesTaxRate table.

Using CROSS Joins

In this example, the Person.StateProvince and Sales.SalesTaxRate tables are CROSS JOIN'd:

```
SELECT   s.CountryRegionCode,
      s.StateProvinceCode,
      t.TaxType,
      t.TaxRate
FROM Person.StateProvince s
CROSS JOIN Sales.SalesTaxRate t
```

This returns the following (abridged) results:

```
CountryRegionCode StateProvinceCode TaxType TaxRate
----------------- ----------------- ------- ---------------------
FR                01                1       14.00
FR                01                1       14.25
FR                01                1       14.25
...
CA                YT                3       19.60
CA                YT                3       17.50

(5249 row(s) affected)
```

How It Works

A CROSS JOIN without a WHERE clause returns a Cartesian product. The results of this CROSS JOIN show StateProvince and SalesTaxRate information that doesn't logically go together. Since the Person. StateProvince table had 181 rows, and the Sales.SalesTaxRate had 29 rows, the query returned 5249 rows.

Performing Self-Joins

Sometimes you may need to treat the same table as two separate tables. This may be because the table contains nested hierarchies of data (for example, employees reporting to managers in the Employees table), or perhaps you wish to reference the same table based on different time periods (compare sales records from the year 1999 versus the year 2005). You can achieve this joining of a table with itself through the use of table aliases.

In this example, a self-join is demonstrated by joining the Employee table's ManagerID with the Employee table's EmployeeID:

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

This returns the following (abridged) results:

```
EmployeeID  Title                                          ManagerTitle
----------- ---------------------------------------------------------
1           Production Technician - WC60                   Production Supervisor - WC60
2           Marketing Assistant                            Marketing Manager
3           Engineering Manager                            Vice President of Engineering
4           Senior Tool Designer                           Engineering Manager
more rows
288         Pacific Sales Manager                          Vice President of Sales
289         Sales Representative                           European Sales Manager
```

```
290         Sales Representative                    Pacific Sales Manager

(290 row(s) affected)
```

How It Works

This example queried the HumanResources.Employee table, returning the EmployeeID of the employee, the employee's title, and the title of his or her manager. The HumanResource.Employee table has a recursive foreign key column called ManagerID, which points to the manager's EmployeeID , and which is the key to another row in the same table. Managers and employees have their data stored in the same table.

Almost all employees have a manager in this table, so using a recursive query, you can establish a nested hierarchy of employees and their managers. There is only one employee that does not have a manager, and that's the Chief Executive Officer.

In the example, the EmployeeID and Title were both taken from the first table aliased with an e. The third column was the title of the Manager, and that table was aliased with an m:

```
SELECT   e.EmployeeID,
      e.Title,
      m.Title AS ManagerTitle
```

The two tables (really the same table, but represented twice using aliases) were joined by EmployeeID to ManagerID using a LEFT OUTER JOIN, so that the Chief Executive Officer would be returned too:

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

Although the same table was referenced twice in the FROM clause, by using a table alias, SQL Server treats them as separate tables.

Tip New to SQL Server 2005, Comment Table Expressions (CTEs) are also reviewed in this chapter, and provide a more sophisticated method of handling recursive queries.

Using Derived Tables

Derived tables are SELECT statements that act as tables in the FROM clause. Derived tables can sometimes provide better performance than using temporary tables (see Chapter 4 for more on temporary tables).

This example demonstrates how to use a derived table in the FROM clause of a SELECT statement:

```
SELECT DISTINCT s.PurchaseOrderNumber
FROM Sales.SalesOrderHeader s
INNER JOIN (SELECT SalesOrderID
         FROM Sales.SalesOrderDetail
         WHERE UnitPrice BETWEEN 1000 AND 2000) d ON
   s.SalesOrderID = d.SalesOrderID
```

This returns the following abridged results:

```
PurchaseOrderNumber
-------------------------
PO8410140860
PO12325137381
PO1160166903
PO1073122178
...
PO15486173227
```

```
PO14268145224

(1989 row(s) affected)
```

How It Works

This example's query searches for the PurchaseOrderNumber from the Sales.SalesOrderHeader table for any order containing products with a UnitPrice between 1000 and 2000.

The query joins a table to a derived table using INNER JOIN. The derived table query is encapsulated in parentheses, and followed by a table alias. The derived table is a separate query in itself, and doesn't require the use of a temporary table to store the results. Thus, queries that use derived tables can sometimes perform significantly better than temporary tables, as you eliminate the steps needed for SQL Server to create and allocate the temporary table prior to use.

Combining Result Sets with UNION

The UNION operator is used to append the results of two or more SELECT statements into a single result set. Each SELECT statement being merged must have the same number of columns, with the same or compatible data types in the same order, as this example demonstrates:

```
SELECT   SalesPersonID, GETDATE() QuotaDate, SalesQuota
FROM      Sales.SalesPerson
WHERE      SalesQuota > 0
UNION
SELECT   SalesPersonID, QuotaDate, SalesQuota
FROM      Sales.SalesPersonQuotaHistory
WHERE      SalesQuota > 0
ORDER BY    SalesPersonID DESC, QuotaDate DESC
```

This returns the following (abridged) results:

```
SalesPersonID QuotaDate               SalesQuota
------------- ----------------------- ---------------------
290           2005-02-27 10:10:12.587 250000.00
290           2004-04-01 00:00:00.000 421000.00
290           2004-01-01 00:00:00.000 399000.00
289           2004-01-01 00:00:00.000 366000.00
289           2003-10-01 00:00:00.000 566000.00
...
268           2002-01-01 00:00:00.000 91000.00
268           2001-10-01 00:00:00.000 7000.00
268           2001-07-01 00:00:00.000 28000.00

(177 row(s) affected)
```

How It Works

This query appended two result sets into a single result set. The first result set returned the SalesPersonID, the current date function (see Chapter 8 for more information on this) and the SalesQuota. Since GETDATE() is a function, it doesn't naturally return a column name—so a QuotaDate column alias was used in its place:

```
SELECT   SalesPersonID, GETDATE() QuotaDate, SalesQuota
FROM      Sales.SalesPerson
```

The WHERE clause filtered data for those salespeople with a SalesQuota greater than zero:

```
WHERE      SalesQuota > 0
```

The next part of the query was the UNION operator, which appended the *distinct* results with the second query:

```
UNION
```

The second query pulled data from the Sales.SalesPersonQuotaHistory, which keeps history for a salesperson's sales quota as it changes through time:

```
SELECT   SalesPersonID, QuotaDate, SalesQuota
FROM      Sales.SalesPersonQuotaHistory
WHERE      SalesQuota > 0
```

The ORDER BY clause sorted the result set by SalesPersonID and QuotaDate, both in descending order. The ORDER BY clause, when needed, must appear at the bottom of the query and cannot appear after queries prior to the final UNION'd query. The ORDER BY clause should also only refer to column names from the *first* result set:

```
ORDER BY SalesPersonID DESC, QuotaDate DESC
```

Looking at the results again, for a single salesperson, you can see that the current QuotaDate of '2005-02-27' is sorted at the top. This was the date retrieved by the GETDATE() function. The other rows for SalesPersonID 290 are from the Sales.SalesPersonQuotaHistory table:

```
SalesPersonID QuotaDate               SalesQuota
------------- ----------------------- ---------------------
290           2005-02-27 10:10:12.587 250000.00
290           2004-04-01 00:00:00.000 421000.00
290           2004-01-01 00:00:00.000 399000.00
290           2003-10-01 00:00:00.000 389000.00
```

Keep in mind that the default behavior of the UNION operator is to remove *all duplicate rows*, and display column names based on the first result set. For large result sets, this can be a very costly operation, so if you don't need to de-duplicate the data, or if the data is naturally distinct, you can add the ALL keyword to the UNION:

```
UNION ALL
```

With the ALL clause added, duplicate rows are NOT removed.

Using APPLY to Invoke a Table-Valued Function for Each Row

New to SQL Server 2005, APPLY is used to invoke a table-valued function for each row of an outer query. A table-valued function returns a result set based on one or more parameters. Using APPLY, the input of these parameters are the columns of the left referencing table. This is useful if the left table contains columns and rows that must be evaluated by the table-valued function.

CROSS APPLY works like an INNER JOIN in that unmatched rows between the left table and the table-valued function don't appear in the result set. OUTER APPLY is like an OUTER JOIN, in that non-matched rows are still returned in the result set with NULL values in the function results.

The next two recipes will demonstrate both CROSS and OUTER APPLY.

Note This next example covers both the FROM and JOIN examples, and user-defined table-valued functions functionality. Table-valued functions are reviewed in more detail in Chapter 11.

Using CROSS APPLY

In this example, a table-valued function is created that returns work order routing information based on the WorkOrderID passed to it:

```
CREATE FUNCTION dbo.fn_WorkOrderRouting
  (@WorkOrderID int) RETURNS TABLE
AS
RETURN
  SELECT WorkOrderID,
       ProductID,
```

```
       OperationSequence,
       LocationID
   FROM Production.WorkOrderRouting
   WHERE WorkOrderID = @WorkOrderID
GO
```

Next, the WorkOrderID is passed from the Production.WorkOrder table to the new function:

```
SELECT   w.WorkOrderID,
      w.OrderQty,
      r.ProductID,
      r.OperationSequence
FROM Production.WorkOrder w
  CROSS APPLY dbo.fn_WorkOrderRouting
  (w.WorkOrderID) AS r
ORDER BY   w.WorkOrderID,
         w.OrderQty,
         r.ProductID
```

This returns the following (abridged) results:

```
WorkOrderID OrderQty    ProductID   OperationSequence
----------- ----------- ----------- -----------------
13          4           747         1
13          4           747         2
13          4           747         3
13          4           747         4
13          4           747         6
13          4           747         7
14          2           748         1
14          2           748         2
...
72585       6           802         1
72585       6           802         6
72586       1           803         1
72586       1           803         6
72587       19          804         1
72587       19          804         6
```

How It Works

The first part of this example was the creation of a table-valued function. The function accepts a single parameter, @WorkOrderID, and when executed, returns the WorkOrderID, ProductID, OperationSequence, and LocationID from the Production.WorkOrderRouting table for the specified WorkOrderID.

The next query in the example returned the WorkOrderID and OrderQty from the Production.WorkOrder table. In addition to this, two columns from the table-valued function were selected:

```
SELECT   w.WorkOrderID,
      w.OrderQty,
      r.ProductID,
      r.OperationSequence
```

The heart of this example comes next. Notice that in the FROM clause, the Production.WorkOrder table is joined to the new table-valued function using CROSS APPLY, only unlike a JOIN clause, there isn't an ON followed by join conditions. Instead, in the parentheses after the function name, the w.WorkOrderID is passed to the table-valued function from the left Production.WorkOrder table:

```
FROM Production.WorkOrder w
  CROSS APPLY dbo.fn_WorkOrderRouting
  (w.WorkOrderID) AS r
```

The function was aliased like a regular table, with the letter "r."

Lastly, the results were sorted:

```
ORDER BY   w.WorkOrderID,
         w.OrderQty,
         r.ProductID
```

In the results for WorkOrderID 13, each associated WorkOrderRouting row was returned next to the calling tables WorkOrderID and OrderQty:

```
WorkOrderID OrderQty    ProductID   OperationSequence
----------- ----------- ----------- -----------------
13          4           747         1
13          4           747         2
13          4           747         3
13          4           747         4
13          4           747         6
13          4           747         7
```

Each row of the WorkOrder table was duplicated for each row returned from fn_WorkOrderRouting—all were based on the WorkOrderID.

Using OUTER APPLY

In order to demonstrate OUTER APPLY, a new row is inserted into Production.WorkOrder:

```
INSERT INTO [AdventureWorks].[Production].[WorkOrder]
           ([ProductID]
           ,[OrderQty]
           ,[ScrappedQty]
           ,[StartDate]
           ,[EndDate]
           ,[DueDate]
           ,[ScrapReasonID]
           ,[ModifiedDate])
     VALUES
           (1,
           1,
           1,
           GETDATE(),
           GETDATE(),
           GETDATE(),
           1,
           GETDATE())
```

Because this is a new row, and because Production.WorkOrder has an IDENTITY column for the WorkOrderID, the new row will have the maximum WorkOrderID in the table. Also, this new row will *not* have an associated value in the Production.WorkOrderRouting table, because it was just added.

Next, a CROSS APPLY query is executed, this time qualifying it to only return data for the newly inserted row:

```
SELECT   w.WorkOrderID,
      w.OrderQty,
      r.ProductID,
      r.OperationSequence
FROM Production.WorkOrder AS w
  CROSS APPLY dbo.fn_WorkOrderRouting
   (w.WorkOrderID) AS r
WHERE w.WorkOrderID IN
      (SELECT MAX(WorkOrderID)
       FROM Production.WorkOrder)
```

This returns nothing, because the left table's new row is unmatched:

```
WorkOrderID OrderQty    ProductID   OperationSequence
----------- ----------- ----------- -----------------

(0 row(s) affected)
```

Now an OUTER APPLY is tried instead, which then returns the row from WorkOrder in spite of there being no associated value in the table-valued function:

```
SELECT   w.WorkOrderID,
      w.OrderQty,
      r.ProductID,
      r.OperationSequence
FROM Production.WorkOrder AS w
  OUTER APPLY dbo.fn_WorkOrderRouting
  (w.WorkOrderID) AS r
WHERE w.WorkOrderID IN
      (SELECT MAX(WorkOrderID)
       FROM Production.WorkOrder)
```

This returns:

```
WorkOrderID OrderQty    ProductID   OperationSequence
----------- ----------- ----------- -----------------
72592       1           NULL        NULL

(1 row(s) affected)
```

How It Works

SQL Sever 2005 has increased the expressiveness of the Transact-SQL language with CROSS and OUTER APPLY, providing a new method for applying lookups against columns, using a table-valued function.

CROSS APPLY was demonstrated against a row without a match in the table-valued function results. Since CROSS APPLY works like an INNER JOIN, no rows were returned. In the second query of this example, OUTER APPLY was used instead, this time returning unmatched NULL rows from the table-valued function, similar to an OUTER JOIN.

Data Source Advanced Techniques

This next set of recipes shows you a few advanced techniques for sampling, manipulating, and comparing data sources (a data source being any valid data source reference in a FROM clause), including:

- Returning a sampling of rows using TABLESAMPLE
- Using PIVOT to convert values into columns, and using an aggregation to group the data by the new columns
- Using UNPIVOT to normalize repeating column groups
- Using INTERSECT and EXCEPT operands to return distinct rows that only exist in either the left query (using EXCEPT), or only distinct rows that exist in both the left and right queries (using INTERSECT)

These recipes all happen to be new features introduced in SQL Server 2005.

Using the TABLESAMPLE to Return Random Rows

Introduced in SQL Server 2005, TABLESAMPLE allows you to extract a sampling of rows from a table in the FROM clause. This sampling can be based on a percentage of number of rows. You

can use TABLESAMPLE when only a sampling of rows is necessary for the application instead of a full result set.

This example demonstrates a query that returns a percentage of random rows from a specific data source using TABLESAMPLE:

```
SELECT FirstName,LastName
FROM Person.Contact TABLESAMPLE SYSTEM (1 PERCENT)
```

This returns the following (abridged) results:

```
FirstName                                          LastName
-------------------------------------------------- --------------------------------
Pat                                                Coleman
Takiko                                             Collins
John                                               Colon
Scott                                              Colvin
more rows
Gerald                                             Lopez
Sydney                                             Johnson
Gerald                                             Martinez

(216 row(s) affected)
```

Executing it again returns a new set of (abridged) results:

```
FirstName                                          LastName
-------------------------------------------------- --------------------------------
Reinout                                            Hillmann
Mike                                               Hines
Matthew                                            Hink
Nancy                                              Hirota
more rows
Jason                                              Ross
Jake                                               Zhang
Jason                                              Coleman

(291 row(s) affected)
```

How It Works

TABLESAMPLE works by extracting a sample of rows from the query result set. In this example, 1 percent of rows were sampled from the Person.Contact table. However don't let the "percent" fool you. That percentage is the percentage of the table's data pages. Once the sample pages are selected, all rows for the selected pages are returned. Since the fill-state of pages can vary, the number of rows returned will also vary—you'll notice that the first time the query is executed in this example there were 216 rows, and the second time there were 291 rows.

If you designate the number of rows, this is actually converted by SQL Server into a percentage, and then the same method used by SQL Server to identify the percentage of data pages is used.

Using PIVOT to Convert Single Column Values into Multiple Columns and Aggregate Data

Introduced in SQL Server 2005, the new PIVOT operator allows you to create cross-tab queries that convert values into columns, using an aggregation to group the data by the new columns.

PIVOT uses the following syntax:

```
FROM  table_source
PIVOT     ( aggregate_function ( value_column )
          FOR pivot_column
          IN ( <column_list>)
    ) table_alias
```

The arguments of PIVOT are described in the Table 1-3.

Table 1-3: PIVOT arguments

Argument	Description
table_source	The table where the data will be pivoted.
aggregate_function (value_column)	The aggregate function that will be used against the specified column.
pivot_column	The column that will be used to create the column headers.
column_list	The values to pivot from the pivot column.
table_alias	The table alias of the pivoted result set.

This next example shows you how to PIVOT and aggregate data similar to Microsoft Excel functionality—such as shifting values in a single column into multiple columns, with aggregated data shown in the results.

The first part of the example displays the data pre-pivoted. The query results show employee shifts, as well as the departments that they are in:

```
SELECT   s.Name ShiftName,
      h.EmployeeID,
      d.Name DepartmentName
FROM   HumanResources.EmployeeDepartmentHistory h
INNER JOIN HumanResources.Department d ON
      h.DepartmentID = d.DepartmentID
INNER JOIN HumanResources.Shift s ON
      h.ShiftID = s.ShiftID
WHERE   EndDate IS NULL AND
      d.Name IN ('Production', 'Engineering', 'Marketing')
ORDER BY ShiftName
```

Notice that the varying departments are all listed in a single column:

```
ShiftName                                          EmployeeID  DepartmentName
-------------------------------------------------- ----------- --------------------
Day                                                3           Engineering
Day                                                9           Engineering
Day                                                11          Engineering
Day                                                12          Engineering
Day                                                267         Engineering
Day                                                270         Engineering
...
Night                                              234         Production
Night                                              245         Production
Night                                              262         Production
Night                                              252         Production

(194 row(s) affected)
```

The next query moves the department values into columns, along with a count of employees by shift:

```
SELECT   ShiftName,
      Production,
      Engineering,
      Marketing
FROM
(SELECT   s.Name ShiftName,
      h.EmployeeID,
      d.Name DepartmentName
FROM   HumanResources.EmployeeDepartmentHistory h
INNER JOIN HumanResources.Department d ON
      h.DepartmentID = d.DepartmentID
INNER JOIN HumanResources.Shift s ON
      h.ShiftID = s.ShiftID
WHERE   EndDate IS NULL AND
      d.Name IN ('Production', 'Engineering', 'Marketing')) AS a
```

```
PIVOT
(
COUNT(EmployeeID)
FOR DepartmentName IN ([Production], [Engineering], [Marketing]))
AS b
ORDER BY ShiftName
```

This returns:

```
ShiftName                                          Production  Engineering Marketing
-------------------------------------------------- ----------- ----------- ---------
Day                                                  79          6           9
Evening                                              54          0           0
Night                                                46          0           0

(3 row(s) affected)
```

How It Works

The result of the PIVOT query returned employee counts by shift and department. The query began by naming the fields to return:

```
SELECT   ShiftName,
      Production,
      Engineering,
      Marketing
```

Notice that these fields were actually the converted rows, but turned into column names.

The FROM clause referenced the subquery (the query used at the beginning of this example. The subquery was aliased with an arbitrary name of “a”:

```
FROM
(SELECT   s.Name ShiftName,
      h.EmployeeID,
      d.Name DepartmentName
FROM   HumanResources.EmployeeDepartmentHistory h
INNER JOIN HumanResources.Department d ON
      h.DepartmentID = d.DepartmentID
INNER JOIN HumanResources.Shift s ON
      h.ShiftID = s.ShiftID
WHERE    EndDate IS NULL AND
      d.Name IN ('Production', 'Engineering', 'Marketing')) AS a
```

Inside the parentheses, the query designated which columns would be aggregated (and how). In this case, the number of employees would be counted:

```
PIVOT
(COUNT(EmployeeID)
```

After the aggregation section, the FOR statement determined which row values would be converted into columns. Unlike regular IN clauses, single quotes aren’t used around each string character, instead using square brackets. DepartmentName was the data column where values are converted into pivoted columns:

```
FOR DepartmentName IN ([Production], [Engineering], [Marketing]))
```

Note The list of column names cannot already exist in the query results being pivoted.

Lastly, a closed parenthesis closed off the PIVOT operation. The PIVOT operation was then aliased like a table with an arbitrary name (in this case "b"):

```
AS b
```

The results were then ordered by ShiftName:

```
ORDER BY ShiftName
```

The results took the three columns fixed in the FOR part of the PIVOT operation and aggregated counts of employees by ShiftName.

Normalizing Data with UNPIVOT

The UNPIVOT command does *almost* the opposite of PIVOT by changing columns into rows. It also uses the same syntax as PIVOT, only UNPIVOT is designated instead.

This example demonstrates how UNPIVOT can be used to remove column-repeating groups often seen in denormalized tables. For the first part of this example, a denormalized table is created with repeating, incrementing phone number columns:

```
CREATE TABLE dbo.Contact
   (EmployeeID int NOT NULL,
    PhoneNumber1 bigint,
    PhoneNumber2 bigint,
    PhoneNumber3 bigint)
GO

INSERT dbo.Contact
(EmployeeID, PhoneNumber1, PhoneNumber2, PhoneNumber3)
VALUES(   1, 2718353881, 3385531980, 5324571342)

INSERT dbo.Contact
(EmployeeID, PhoneNumber1, PhoneNumber2, PhoneNumber3)
VALUES(   2, 6007163571, 6875099415, 7756620787)

INSERT dbo.Contact
(EmployeeID, PhoneNumber1, PhoneNumber2, PhoneNumber3)
VALUES(   3, 9439250939, NULL, NULL)
```

Now using UNPIVOT, the repeating phone numbers are converted into a more normalized form (re-using a single PhoneValue field instead of repeating the phone column multiple times):

```
SELECT   EmployeeID,
      PhoneType,
      PhoneValue
FROM
(SELECT   EmployeeID, PhoneNumber1, PhoneNumber2, PhoneNumber3
FROM dbo.Contact) c
UNPIVOT
   (PhoneValue FOR PhoneType IN ([PhoneNumber1], [PhoneNumber2], [PhoneNumber3])
) AS p
```

This returns:

```
EmployeeID  PhoneType         PhoneValue
----------- ----------------- ----------
1           PhoneNumber1       2718353881
1           PhoneNumber2       3385531980
1           PhoneNumber3       5324571342
2           PhoneNumber1       6007163571
2           PhoneNumber2       6875099415
2           PhoneNumber3       7756620787
3           PhoneNumber1       9439250939
```

```
(7 row(s) affected)
```

How It Works

This UNPIVOT example began by selecting three columns. The EmployeeID came from the subquery. The other two columns, PhoneType and PhoneValue—were defined later on in the UNPIVOT statement:

```
SELECT   EmployeeID,
      PhoneType,
      PhoneValue
```

Next, the FROM clause referenced a subquery. The subquery selected all four columns from the contact table. The table was aliased with the letter "c" (table alias naming was arbitrary however):

```
FROM
(SELECT    EmployeeID, PhoneNumber1, PhoneNumber2, PhoneNumber3
FROM dbo.Contact) c
```

A new column called PhoneValue (referenced in the SELECT) holds the individual phone numbers across the three denormalized phone columns:

```
UNPIVOT
(PhoneValue FOR PhoneType IN ([PhoneNumber1], [PhoneNumber2], [PhoneNumber3])
```

FOR references the name of the pivot column, PhoneType, which holds the column names of the denormalized table. The IN clause following PhoneType lists the columns from the original table to be narrowed into a single column.

Lastly, a closing parenthesis is used, then alias it with an arbitrary name, in this case "p":

```
) AS p
```

This query returned the phone data merged into two columns, one to describe the phone type, and another to hold the actual phone numbers. Also notice that there are seven rows, instead of nine. This is because for EmployeeID "3", only non-NULL values were returned. UNPIVOT does not process NULL values from the pivoted result set.

Returning Distinct or Matching Rows Using EXCEPT and INTERSECT

Introduced in SQL Server 2005, the INTERSECT and EXCEPT operands allow you to return distinct rows that only exist in either the left query (using EXCEPT), or only distinct rows that exist in both the left and right queries (using INTERSECT).

INTERSECT and EXCEPT are useful in dataset comparison scenarios; for example, if you need to compare rows between test and production tables, you can use EXCEPT to easily identify and populate rows that existed in one table and not the other. These operands are also useful for data recovery, because you could restore a database from a period prior to a data loss, compare data with the current production table, and then recover the deleted rows accordingly.

For this example, demonstration tables are created which are partially populated from the Production.Product table:

```
-- First two new tables based on ProductionProduct will be
-- created, in order to demonstrate EXCEPT and INTERSECT.
-- See Chapter 8 for more on ROW_NUMBER

-- Create TableA
SELECT   prod.ProductID,
      prod.Name
INTO dbo.TableA
FROM
(SELECT ProductID,
      Name,
```

```
      ROW_NUMBER() OVER (ORDER BY ProductID) RowNum
FROM Production.Product) prod
WHERE RowNum BETWEEN 1 and 20

-- Create TableB
SELECT   prod.ProductID,
      prod.Name
INTO dbo.TableB
FROM
(SELECT ProductID,
      Name,
      ROW_NUMBER() OVER (ORDER BY ProductID) RowNum
FROM Production.Product) prod
WHERE RowNum BETWEEN 10 and 29
```

This returns:

```
(20 row(s) affected)

(20 row(s) affected)
```

Now the EXCEPT operator will be used to determine which rows exist *only* in the left table of the query, TableA, and not in TableB:

```
SELECT   ProductID,
      Name
FROM   TableA
EXCEPT
SELECT   ProductID,
      Name
FROM   TableB
```

This returns:

```
ProductID   Name
----------- --------------------------------------------------
1           Adjustable Race
2           Bearing Ball
3           BB Ball Bearing
4           Headset Ball Bearings
316         Blade
317         LL Crankarm
318         ML Crankarm
319         HL Crankarm
320         Chainring Bolts

(9 row(s) affected)
```

To show distinct values from *both* result sets that match, use the INTERSECT operator:

```
SELECT   ProductID,
      Name
FROM   TableA
INTERSECT
SELECT   ProductID,
      Name
FROM   TableB
```

This returns:

```
ProductID   Name
----------- --------------------------------------------------
321         Chainring Nut
322         Chainring
323         Crown Race
324         Chain Stays
325         Decal 1
326         Decal 2
327         Down Tube
328         Mountain End Caps
```

```
329         Road End Caps
330         Touring End Caps
331         Fork End

(11 row(s) affected)
```

How It Works

The example started off by creating two tables (using INTO) that contain overlapping sets of rows. The first table, TableA, contained the first twenty rows (ordered by ProductID) from the Production. Product table. The second table, TableB, also received another twenty rows, half of which overlapped with TableA's rows.

To determine which rows exist *only* in TableA, the EXCEPT operand was placed after the FROM clause of the first query and before the second query:

```
SELECT   ProductID,
      Name
FROM   TableA
EXCEPT
SELECT   ProductID,
      Name
FROM   TableB
```

In order for EXCEPT to be used, both queries must have the same number of columns. Those columns also need to have compatible data types (it's not necessary that the column names from each query match). The power of EXCEPT is that *all* columns are evaluated to determine if there is a match, which is much more efficient than using INNER JOIN (which would require explicitly joining the tables on each column in both data sources).

The results of the EXCEPT query show the first nine rows from TableA that were not also populated into TableB.

In the second example, INTERSECT was used to shows rows that *overlap* between the two tables. Like EXCEPT, INTERSECT is placed between the two queries:

```
SELECT   ProductID,
      Name
FROM   TableA
INTERSECT
SELECT   ProductID,
      Name
FROM   TableB
```

The query returned the eleven rows that overlapped between both tables. The same rules about compatible data types and number of columns apply to INTERSECT as for EXCEPT.

Summarizing Data

In these next three recipes, I will demonstrate summarizing data within the result set using the following operators:

- Use WITH CUBE to add summarizing total values to a result-set based on columns in the GROUP BY clause.

- Use WITH ROLLUP with GROUP BY to add hierarchical data summaries based on the ordering of columns in the GROUP BY clause.

I also demonstrate the GROUPING function. GROUPING is used to determine which of these summarized total value rows are based on the summarized or original data set data.

Summarizing Data with WITH CUBE

WITH CUBE adds rows to your result-set, summarizing total values based on the columns in the GROUP BY clause.

This example demonstrates a query that returns the total quantity of a product, grouped by the shelf the product is kept on:

```
SELECT   i.Shelf,
      SUM(i.Quantity) Total
FROM   Production.ProductInventory i
GROUP BY i.Shelf
WITH CUBE
```

This returns the following (abridged) results:

```
Shelf      Total
---------- -----------
A          26833
B          12672
C          19868
D          17353
E          31979
F          21249
G          40195
H          20055
more rows
W          2908
Y          437
NULL       335974
```

How It Works

Because the query in this example groups by Shelf, a total will be displayed, displaying the total for all shelves in the final row. With WITH CUBE added after the GROUP BY clause, an extra row with a NULL Shelf value is added at the end of the results, along with the SUM total of all quantities in the Total column.

If you added additional columns to the query, included in the GROUP BY clause, WITH CUBE would attempt to aggregate values for each grouping combination. WITH CUBE is often used for reporting purposes, providing an easy method of reporting totals by grouped column.

Note In SQL Server 2000, you may have used COMPUTE BY to also provide similar aggregations for your query. Microsoft has deprecated this functionality for SQL Server 2005 backward compatibility. Unlike WITH CUBE, COMPUTE BY created an entirely new summarized result set after the original query results which were often difficult for calling applications to consume.

Using GROUPING with WITH CUBE

In the previous example, WITH CUBE was used to aggregate values for each grouping combination. Extra NULL values were included in the result set for those rows that contained the WITH CUBE aggregate totals.

What if one of the values in the SHELF column was actually NULL? In order to distinguish between a NULL that comes from the source data versus a NULL generated by a WITH CUBE aggregation, you can use the GROUPING function. This function returns a “0” value when the data is derived from the data, and a “1” when generated by a WITH CUBE.

This example modifies the previous recipe’s example to include GROUPING:

```
SELECT   i.Shelf,
   GROUPING(i.Shelf) Source,
   SUM(i.Quantity) Total
```

```
FROM   Production.ProductInventory i
GROUP BY i.Shelf
WITH CUBE
```

This returns the following (abridged) results:

```
Shelf      Source Total
---------- ------ -----------
A          0      26833
B          0      12672
Y          0      437
NULL       1      335974
```

How It Works

In this recipe, GROUPING was used to discern a natural NULL versus a NULL generated by the WITH CUBE function. You can also use the GROUPING function with the ROLLUP function, which is reviewed next.

Summarizing Data with WITH ROLLUP

WITH ROLLUP is used in conjunction with GROUP BY to add hierarchical data summaries based on the ordering of columns in the GROUP BY clause.

This example retrieves the shelf, product name, and total quantity of the product:

```
SELECT   i.Shelf,
      p.Name,
      SUM(i.Quantity) Total
FROM   Production.ProductInventory i
INNER JOIN Production.Product p ON
   i.ProductID = p.ProductID
GROUP BY i.Shelf, p.Name
WITH ROLLUP
```

This returns the following (abridged) results:

```
Shelf      Name                                               Total
---------- -------------------------------------------------- -----------
A          Adjustable Race                                    761
A          BB Ball Bearing                                    909
A          Bearing Ball                                       791
...
A          NULL                                               26833
...
W          Rear Derailleur Cage                               121
W          Reflector                                          121
W          Touring Pedal                                      388
W          NULL                                               2908
Y          HL Spindle/Axle                                    228
Y          LL Spindle/Axle                                    209
Y          NULL                                               437
NULL       NULL                                               335974
```

How It Works

The order you place the columns in the GROUP BY impacts how WITH ROLLUP aggregates the data. The WITH ROLLUP in this query aggregated total quantity for each change in Shelf. Notice the row with Shelf "A" and the NULL name; this holds the total quantity for Shelf A. Also notice that the final row was the grand total of all product quantities.

Hints

SQL Server's query optimization process is responsible for producing a query execution plan when a SELECT query is executed. The goal of the query optimizer is to generate an efficient (but not always the best) query execution plan. Under rare circumstances SQL Server may choose an inefficient plan over a more efficient one. If this happens you would be advised to investigate the query execution plan, table statistics, and other factors that are explored in more detail in Chapter 28.

After researching the query performance, you may decide to override the decision making process of the SQL Server query optimizer by using *hints*. The next three recipes will demonstrate three types of hints: *query*, *table*, and *join*.

Using Join Hints

A join "hint" is a misnomer in this case, as a join hint will *force* the query optimizer to join the tables in the way you command. Join hints force the internal JOIN operation used to join two tables in a query. Available join hints are described in Table 1-4.

Table 1-4: Join Hints

Hint Name	Description
LOOP	LOOP joins operate best when one table is small and the other is large, with indexes on the joined columns.
HASH	HASH joins are optimal for large unsorted tables.
MERGE	MERGE joins are optimal for medium or large tables that are sorted on the joined column.
REMOTE	REMOTE forces the join operation to occur at the site of the table referenced on the right (the second table referenced in a JOIN clause). For performance benefits, the left table should be the local table, and should have fewer rows than the remote right table.

Before showing how the join hint works, the example starts off with the original, non-hinted query:

```
-- (More on SHOWPLAN_TEXT in Chapter 28)
SET SHOWPLAN_TEXT ON
GO

SELECT   p.Name,
      r.ReviewerName,
      r.Rating
FROM   Production.Product p
INNER JOIN Production.ProductReview r ON
   r.ProductID = p.ProductID
GO

SET SHOWPLAN_TEXT OFF
GO
```

This returns the following abridged results (SHOWPLAN_TEXT returns information about how the query *may be* processed, but it doesn't actually execute the query):

```
StmtText

------------------------------------------------------------------------------------
  |--Nested Loops(Inner Join, OUTER REFERENCES:([r].[ProductID]))
       |--Clustered Index
Scan(OBJECT:([AdventureWorks].[Production].[ProductReview].[PK_ProductReview_
ProductReviewID] AS [r]))
       |--Clustered Index
Seek(OBJECT:([AdventureWorks].[Production].[Product].[PK_Product_ProductID]
AS [p]), SEEK:([p].[ProductID]=[AdventureWorks].[Production].[ProductReview].
```

```
[ProductID] as [r].[ProductID]) ORDERED FORWARD)
```

The next example submits the same query, only this time using a join hint:

```
SET SHOWPLAN_TEXT ON
GO

SELECT   p.Name,
      r.ReviewerName,
      r.Rating
FROM   Production.Product p
INNER HASH JOIN Production.ProductReview r ON
   r.ProductID = p.ProductID
GO

SET SHOWPLAN_TEXT OFF
GO
```

This returns the following abridged results:

```
StmtText
-----------------------------------------------------------------------------------
|--Hash Match(Inner Join, HASH:([p].[ProductID])=([r].[ProductID]))
       |--Index Scan(OBJECT:([AdventureWorks].[Production].[Product].
[AK_Product_Name] AS [p]))
       |--Clustered Index Scan(OBJECT:([AdventureWorks].[Production].
[ProductReview].
[PK_ProductReview_ProductReviewID] AS [r]))
```

How It Works

In the first, non-hinted query, SET SHOWPLAN_TEXT was used to view how the query may be executed by SQL Server.

Caution You should almost always let SQL Server do the decision-making for the join type. Even if your hint works for the short term, there is no guarantee that in the future there may be more efficient query plans that could be used, but won't be, because you have overridden the optimizer with the specified hint. Also, the validity of or effectiveness of a hint may change when new service packs or editions of SQL Server are released.

The StmtText section, shown next, tells us how the query will be executed. For this example, the most important information was in the first line of the results, telling us that a Nested Loop would be used to join the two tables:

```
|--Nested Loops(Inner Join, OUTER REFERENCES:([r].[ProductID]))
```

In the second query, a hint was added to force the nested loop join to perform a hash join operation instead. To do this, HASH was added between the INNER and JOIN keywords:

```
INNER HASH JOIN Production.ProductReview r ON
   r.ProductID = p.ProductID
```

Now in the second SHOWPLAN_TEXT results, the query execution now uses a hash join to join the two tables:

```
|--Hash Match(Inner Join, HASH:([p].[ProductID])=([r].[ProductID]))
```

Using Query Hints

Some query hints, like the join hints discussed in the previous recipe, are instructions sent with the query to override SQL Server's query optimizer decision-making. Using query hints may provide a short term result that satisfies your current situation, but may not always be the most efficient result

over time. Nonetheless, there are times when you may decide to use them, if only to further understand the choices that the query optimizer automatically makes.

Query hints can be used in SELECT, INSERT, UPDATE, and DELETE statements, described in Table 1-5.

Table 1-5: Query Hints

Hint Name	Description
{HASH | ORDER} GROUP	When used in conjunction with the GROUP BY clause, specifies whether hashing or ordering is used for GROUP BY and COMPUTE aggregations.
{CONCAT | HASH | MERGE} UNION	Designates the strategy used to join all result sets for UNION operations.
{LOOP | MERGE | HASH} JOIN	Forces *all* join operations to perform the loop, merge, or hash join in the entire query.
FAST integer	Speeds up the retrieval of rows for the top *integer* value designated.
FORCE ORDER	When designated, table joins are performed in the order in which the tables appear.
MAXDOP number_of_processors	This option overrides the "max degree of parallelism" server configuration option for the query. See Chapter 21 for more information on this server option.
OPTIMIZE FOR (@variable_name = literal_constant) [,...n]	The OPTIMIZE FOR option directs SQL Server to use a particular variable value or values for a variable when the query is compiled and optimized . You could, for example, plug in a literal constant that returns the best performance across the range of expected parameters.
ROBUST PLAN	Creates a query plan with the assumption that the row size of a table will be at maximum width.
MAXDOP integer	This determines the max degree of parallelism for your query (overriding the server level setting).
KEEP PLAN	The recompile threshold for the query is "relaxed" when this hint is used.
KEEPFIXED PLAN	When this hint is used, the query optimizer is forced NOT to recompile due to statistics or indexed column changes. Only schema changes or sp_recompile will cause the query plan to be recompiled.
EXPAND VIEWS	This hint keeps the query optimizer from using indexed views when the base table is referenced.
MAXRECURSION number	Designates the maximum number of recursions (1 to 32757) allowed for the query. If 0 is chosen, no limit is applied. The default recursion limit is 100. This option is used in conjunction with Common Table Expressions (CTE).
USE PLAN 'xml_plan'	USE PLAN directs SQL SERVER to use a potentially better performing query plan (provided in the xml_plan literal value) that you know can cause the query to perform better. See Chapter 28 for more detail.
PARAMETERIZATION { SIMPLE | FORCED }	This hint relates to the new PARAMETERIZATION database setting which controls whether or not all queries are parameterized (literal values contained in a query get substituted with parameters in the cached query plan). When PARAMETERIZATION SIMPLE is chosen, SQL Server decides which queries are parameterized or not. When PARAMETERIZATION FORCED is used, all queries in the database will be parameterized. For more information on this database setting, see Chapter 28.
RECOMPILE	This forces SQL Server 2005 to throw out the query execution plan after it is executed, meaning that the next time the query executes, it will be forced to recompile a new query plan. Although usually SQL Server re-uses effective query plans—sometimes a less efficient query

	plan is re-used. Recompiling forces SQL Server to come up with a fresh plan (but with the overhead of a recompile).

This example uses a new SQL Server 2005 RECOMPILE query hint to recompile the query, forcing SQL Server to discard the plan generated for the query after it executes. With the RECOMPILE query hint, a new plan will be generated the next time the same or similar query is executed. You may decide you wish to do this for volatile query plans, where differing search condition values for the same plan cause extreme fluctuations in the number of rows returned. In that scenario, using a compiled query plan may hurt, not help the query performance. The benefit of a cached and reusable query execution plan may occasionally be outweighed by a fresh, recompiled plan.

Note SQL Server 2005 has introduced statement-level stored procedure recompilation. Now instead of an entire stored procedure recompiling when indexes are added or data is changed to the referenced tables, only individual statements within the procedure impacted by the change are recompiled. See Chapter 10 for more information.

Typically, you will want to use this RECOMPILE query hint within a stored procedure—so that you can control which statements automatically recompile—instead of having to recompile the entire stored procedure. Now for the example:

```
SELECT   SalesOrderID,
   ProductID,
   UnitPrice,
   OrderQty
FROM Sales.SalesOrderDetail
WHERE CarrierTrackingNumber = '5CE9-4D75-8F'
ORDER BY   SalesOrderID,
         ProductID
OPTION (RECOMPILE)
```

This returns:

```
SalesOrderID ProductID   UnitPrice             OrderQty
------------ ----------- --------------------- --------
47964        760         469.794               1
47964        789         1466.01               1
47964        819         149.031               4
47964        843         15.00                 1
47964        844         11.994                6
(5 row(s) affected)
```

How It Works

This query demonstrated using a query hint, which was referenced in the OPTION clause at the end of the query:

```
OPTION (RECOMPILE)
```

SQL Server 2005 should be relied upon most of the time to make the correct decisions when processing a query; however query hints provide you with more control for those exceptions when you need to override SQL Server's choices.

Using Table Hints

Table hints, like query hints, can be used to override SELECT, INSERT, UPDATE, and DELETE default processing behavior. You can use multiple table hints for one query, separated by commas, so long as they do not belong to the same category grouping. Be sure to test the performance of your queries with and without the query hints (see Chapter 28 for more details on examining query performance).

Table 1-6 lists available table hints. Some hints cannot be used together, so they have been grouped in the table accordingly. You can't, for example, use both NOLOCK and HOLDLOCK for the same query:

Table 1-6: Table Hints

Hint Name	Description
FASTFIRSTROW	This hint optimizes the query to pull the first row of the result set very quickly. Whether or not this hint will work depends on the size of your table, indexing used, and the type of data you are returning. Test your results with and without the hint, to be sure that the hint is necessary. Use this hint in order to begin returning results to the client faster – not to improve the speed of the entire result set.
INDEX (index_val [,... n])	Overrides SQL Server's index choice and forces a specific index for the table to be used.
NOEXPAND	When an indexed view is referenced, the query optimizer will treat the view like a table with a clustered index.
HOLDLOCK. SERIALIZABLE. REPEATABLEREAD. READCOMMITTED. READCOMMITTEDLOCK. READUNCOMMITTED. NOLOCK	Selecting one of these hints determines the isolation level for the table. For example, designating NOLOCK means that the operation (SELECT for example) will place no locking on the table.
ROWLOCK. PAGLOCK. TABLOCK. TABLOCKX. NOLOCK	Designates the granularity of locking for the table, for example, selecting ROWLOCK to force only row locks for a query.
READPAST	Skips locked rows, and does not read them.
UPDLOCK	The hint will force update locks instead of shared locks to be generated (not compatible with NOLOCK or XLOCK).
XLOCK	This hint forces exclusive locks on the resources being referenced (not compatible with NOLOCK or UPDLOCK).
KEEPIDENTITY	This option applies to the OPENROWSET function's new SQL Server 2005 BULK insert functionality (see Chapter 27) and impacts how rows are inserted into a table with an IDENTITY column. If you use this hint, SQL Server will use the identity values from the data file, instead of generating its own. For more on the IDENTITY column, see Chapter 4, "Tables."
KEEPDEFAULTS	Like KEEPIDENTITY, this table hint applies to the OPENROWSET function. Using this hint specifies that columns not included in the bulk-load operation will be assigned to the column default. For more on default columns, see Chapter 4.
IGNORE_CONSTRAINTS	Another OPENROWSET hint, IGNORE_CONSTRAINTS directs SQL Server to ignore CHECK constraints when importing data. See Chapter 4 for more on CHECK constraints.
IGNORE_TRIGGERS	

	This query hint directs INSERT triggers *not* to fire when importing using the new SQL Server 2005 BULK option of OPENROWSET.

This example returns the DocumentID and Title from the Production.Document table where the Status column is equal to "1." It uses the NOLOCK table hint, which means the query will not place shared locks on the Production.Document table (for a review of locking, see Chapter 3):

```
SELECT DocumentID,
      Title
FROM Production.Document
WITH (NOLOCK)
WHERE Status = 1
```

How It Works

The crux of this example is the WITH keyword, which uses the NOLOCK table hint in parentheses:

```
WITH (NOLOCK)
```

NOLOCK causes the query not to place shared locks on the impacted rows/data pages—allowing you to read without being blocked or blocking others (although you are now subject to "dirty reads").

Common Table Expressions

A Common Table Expression (CTE) is similar to a view or derived query, allowing you to create a temporary query that can be referenced within the scope of a SELECT, INSERT, UPDATE, or DELETE query. Unlike a derived query, you don't need to copy the query definition multiple times each time it is used. You can also use local variables within a CTE definition—something you can't do in a view definition.

The basic syntax for a CTE is as follows:

```
WITH expression_name [ ( column_name [ ,...n ] ) ]
AS ( CTE_query_definition )
```

The arguments of a CTE are described in the Table 1-7.

Table 1-7: CTE Arguments

Argument	Description
expression_name	The name of the common table expression.
column_name [,...n]	The unique column names of the expression.
CTE_query_definition	The SELECT query that defines the common table expression.

A *non-recursive* CTE is one that is used within a query without referencing itself. It serves as a temporary result set for the query. A *recursive* CTE is defined similarly to a non-recursive CTE, only a recursive CTE returns hierarchical self-relating data. Using a CTE to represent recursive data can minimize the amount of code needed compared to other methods.

The next two recipes will demonstrate both non-recursive and recursive CTEs.

Using a Non-Recursive Common Table Expression (CTE)

This example of a common table expression demonstrates returning vendors in the Purchasing.Vendor table—returning the first five and last five results ordered by name:

```
WITH VendorSearch (RowNumber, VendorName, AccountNumber)
AS
(
SELECT ROW_NUMBER() OVER (ORDER BY Name) RowNum,
      Name,
      AccountNumber
FROM Purchasing.Vendor
)

SELECT   RowNumber,
      VendorName,
      AccountNumber
FROM VendorSearch
WHERE RowNumber BETWEEN 1 AND 5
UNION
SELECT   RowNumber,
      VendorName,
      AccountNumber
FROM VendorSearch
WHERE RowNumber BETWEEN 100 AND 104
```

This returns:

```
RowNumber            VendorName                                         AccountNumber
-------------------- -------------------------------------------------- ------------
1                    A. Datum Corporation                               ADATUM0001
2                    Advanced Bicycles                                  ADVANCED0001
3                    Allenson Cycles                                    ALLENSON0001
4                    American Bicycles and Wheels                       AMERICAN0001
5                    American Bikes                                     AMERICAN0002
100                  Vista Road Bikes                                   VISTARO0001
101                  West Junction Cycles                               WESTJUN0001
102                  WestAmerica Bicycle Co.                            WESTAMER0001
103                  Wide World Importers                               WIDEWOR0001
104                  Wood Fitness                                       WOODFIT0001

(10 row(s) affected)
```

The previous example used UNION, however non-recursive CTEs can be used like any other SELECT query too:

```
WITH VendorSearch (VendorID, VendorName)
AS
(
SELECT VendorID,
      Name
FROM Purchasing.Vendor
)

SELECT   v.VendorID,
      v.VendorName,
      p.ProductID,
      p.StandardPrice
FROM VendorSearch v
INNER JOIN Purchasing.ProductVendor p ON
   v.VendorID = p.VendorID
ORDER BY v.VendorName
```

This returns the following (abridged) results:

```
VendorID    VendorName                                         ProductID   StandardPrice
----------- -------------------------------------------------- ----------- ---------
32          Advanced Bicycles                                      359         45.41
32          Advanced Bicycles                                      360         43.41
32          Advanced Bicycles                                      361         47.48
...
91          WestAmerica Bicycle Co.                                363         41.26
28          Wide World Importers                                   402         45.21
57          Wood Fitness                                           2           39.92

(406 row(s) affected)
```

How It Works

In this example, WITH defined the CTE name and the columns it returned. This was a non-recursive CTE because CTE data wasn't being joined to itself. The CTE in this example was only using a query that UNION'd two data sets:

```
WITH VendorSearch (RowNumber, VendorName, AccountNumber)
```

The column names defined in the CTE can match the actual names of the query within—or you can create your own alias names. For example in this example, the Purchasing.Vendor column Name has been referenced as VendorName in the CTE.

Next in the recipe, AS marked the beginning of the CTE query definition:

```
AS
(
```

Inside the parentheses, the query used a new SQL Server 2005 function that returned the sequential row number of the result set—ordered by the vendor name (see Chapter 8 for a review of ROW_NUMBER):

```
SELECT ROW_NUMBER() OVER (ORDER BY Name) RowNum,
      Name,
      AccountNumber
FROM Purchasing.Vendor)
```

The Vendor Name and AccountNumber from the Purchasing.Vendor table were also returned. The CTE definition finished after marking the closing parentheses.

Following the CTE definition was the query that used the CTE. Keep in mind that a SELECT, INSERT, UPDATE, or DELETE statement that references some or all the CTE columns *must* follow the definition of the CTE:

```
SELECT   RowNumber,
   VendorName,
   AccountNumber
FROM VendorSearch
WHERE RowNumber BETWEEN 1 AND 5
```

The SELECT column names were used from the new VendorSearch CTE. In the WHERE clause, the first query returns rows 1 through 5. Next the UNION operator was used prior to the second query:

```
UNION
```

This second query displayed the last five rows in addition to the first five rows. The VendorSearch CTE was referenced twice—but the full query definition only had to be defined a single time (unlike derived queries)—thus reducing code.

In the second example of the recipe, a simple CTE was defined without using any functions, just VendorID and VendorName from the Purchasing.Vendor table:

```
WITH VendorSearch (VendorID, VendorName)
AS
(
SELECT VendorID,
      Name
FROM Purchasing.Vendor)
```

In the query following this CTE definition, the CTE "VendorSearch" was joined just like a regular table (only without specifying the owning schema):

```
SELECT   v.VendorID,
      v.VendorName,
      p.ProductID,
```

```
      p.StandardPrice
FROM VendorSearch v
INNER JOIN Purchasing.ProductVendor p ON
   v.VendorID = p.VendorID
ORDER BY v.VendorName
```

One gotcha that you should be aware of—if the CTE is part of a batch of statements, the statement before its definition must be followed by a *semicolon*.

Note With SQL Server 2005, you can use a semicolon as a SQL Server statement terminator. Doing so isn't mandatory, but is ANSI compliant, and you'll see it being used in some of the documentation coming from Microsoft.

Using a Recursive Common Table Expression (CTE)

In this example, the new Company table will define the companies in a hypothetical giant mega conglomerate. Each company has a CompanyID and an optional ParentCompanyID. The example will demonstrate how to display the company hierarchy in the results using a recursive CTE. First, the table is created:

```
CREATE TABLE dbo.Company
   (CompanyID int NOT NULL PRIMARY KEY,
    ParentCompanyID int NULL,
    CompanyName varchar(25) NOT NULL)
```

Next, rows are inserted into the new table:

```
INSERT dbo.Company (CompanyID, ParentCompanyID, CompanyName)
VALUES (1, NULL, 'Mega-Corp')
INSERT dbo.Company (CompanyID, ParentCompanyID, CompanyName)
VALUES (2, 1, 'Mediamus-Corp')
INSERT dbo.Company (CompanyID, ParentCompanyID, CompanyName)
VALUES (3, 1, 'KindaBigus-Corp')
INSERT dbo.Company (CompanyID, ParentCompanyID, CompanyName)
VALUES (4, 3, 'GettinSmaller-Corp')
INSERT dbo.Company (CompanyID, ParentCompanyID, CompanyName)
VALUES (5, 4, 'Smallest-Corp')
INSERT dbo.Company (CompanyID, ParentCompanyID, CompanyName)
VALUES (6, 5, 'Puny-Corp')
INSERT dbo.Company (CompanyID, ParentCompanyID, CompanyName)
VALUES (7, 5, 'Small2-Corp')
```

Now the actual example:

```
WITH CompanyTree(ParentCompanyID, CompanyID, CompanyName, CompanyLevel)
AS
(
    SELECT    ParentCompanyID,
      CompanyID,
      CompanyName,
      0 AS CompanyLevel
    FROM dbo.Company
    WHERE ParentCompanyID IS NULL
    UNION ALL
    SELECT    c.ParentCompanyID,
      c.CompanyID,
      c.CompanyName,
      p.CompanyLevel + 1
    FROM dbo.Company c
        INNER JOIN CompanyTree p
        ON c.ParentCompanyID = p.CompanyID
)
SELECT ParentCompanyID, CompanyID, CompanyName, CompanyLevel
FROM CompanyTree
```

This returns:

```
ParentCompanyID CompanyID   CompanyName               CompanyLevel
--------------- ----------- ------------------------- ------------
NULL            1           Mega-Corp                 0
1               2           Mediamus-Corp             1
1               3           KindaBigus-Corp           1
3               4           GettinSmaller-Corp        2
4               5           Smallest-Corp             3
5               7           Small2-Corp               4
5               6           Puny-Corp                 4

(7 row(s) affected)
```

How It Works

In this example, the CTE name and columns are defined first:

```
WITH CompanyTree(ParentCompanyID, CompanyID, CompanyName, CompanyLevel)
```

The CTE query definition began with AS and an open parenthesis:

```
AS
(
```

The SELECT clause began with the "anchor" SELECT statement. When using recursive CTEs, "anchor" refers to the fact that it defines the base of the recursion—in this case the top level of the corporate hierarchy (the parentless "Mega-Corp"). This SELECT also includes a CompanyLevel column alias, preceded with the number zero. This column will be used in the recursion to display how many levels deep a particular company is in the company hierarchy:

```
SELECT ParentCompanyID,
  CompanyID,
  CompanyName,
  0 AS CompanyLevel
FROM dbo.Company
WHERE ParentCompanyID IS NULL
```

Next was the UNION ALL, to join the second, recursive query to the anchor member (UNION ALL, and not just UNION, is required for the last anchor member and the first recursive member in a recursive CTE):

```
UNION ALL
```

After that was the recursive query. Like the anchor, the SELECT clause references the ParentCompanyID, CompanyID, and CompanyName from the dbo.Company table. Unlike the anchor, the CTE column references p.CompanyLevel (from the anchor query), adding + 1 to its total at each level of the hierarchy:

```
SELECT    c.ParentCompanyID,
  c.CompanyID,
  c.CompanyName,
  p.CompanyLevel + 1
```

The dbo.Company table was joined to the CompanyTree CTE, joining the CTE's recursive query's ParentCompanyID to the CTE's CompanyID:

```
    FROM dbo.Company c
        INNER JOIN CompanyTree p
        ON c.ParentCompanyID = p.CompanyID
)
```

After the closing of the CTE's definition, the query selected from the CTE based on the columns defined in the CTE definition.

```
SELECT ParentCompanyID, CompanyID, CompanyName, CompanyLevel
FROM CompanyTree
```

In the results, for each level in the company hierarchy, the CTE increased the CompanyLevel column.

With this useful new feature come some cautions, however. If you create your recursive CTE incorrectly, you could cause an infinite loop. While testing, and to avoid infinite loops, use the MAXRECURSION hint mentioned earlier in the chapter.

For example, you can stop the previous example from going further than 2 levels by adding the OPTION clause with MAXRECURSION at the end of the query:

```
WITH CompanyTree(ParentCompanyID, CompanyID, CompanyName, CompanyLevel) AS
(
    SELECT ParentCompanyID, CompanyID, CompanyName, 0 AS CompanyLevel
    FROM dbo.Company
    WHERE ParentCompanyID IS NULL
    UNION ALL
    SELECT c.ParentCompanyID, c.CompanyID, c.CompanyName, p.CompanyLevel + 1
    FROM dbo.Company c
        INNER JOIN CompanyTree p
        ON c.ParentCompanyID = p.CompanyID
)
SELECT ParentCompanyID, CompanyID, CompanyName, CompanyLevel
FROM CompanyTree
OPTION (MAXRECURSION 2)
```

This returns:

```
ParentCompanyID CompanyID   CompanyName               CompanyLevel
--------------- ----------- ------------------------- ------------
NULL            1           Mega-Corp                 0
1               2           Mediamus-Corp             1
1               3           KindaBigus-Corp           1
3               4           GettinSmaller-Corp        2
Msg 530, Level 16, State 1, Line 1
Statement terminated. Maximum recursion 2 has been exhausted before statement completion
```

As a best practice, set the MAXRECURSION based on your understanding of the data. If you know that the hiearchy cannot go more than 10 levels deep, for example, then set MAXRECURSION to that value.