TRAINING & REFERENCE

murach's SQL Server 2008 for developers

(Chapter 3)

Thanks for reviewing this chapter from ***Murach's SQL Server 2008 for Developers***. To see the expanded table of contents for this book, you can go to the Murach web site. From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review other Murach books for professional developers.

MIKE MURACH & ASSOCIATES, INC.
1-800-221-5528 • (559) 440-9071 • Fax: (559) 440-0963
murachbooks@murach.com • www.murach.com

Section 2

The essential SQL skills

This section teaches you the essential SQL coding skills for working with the data in a SQL Server database. The first four chapters in this section show you how to retrieve data from a database using the SELECT statement. In chapter 3, you'll learn how to code the basic clauses of the SELECT statement to retrieve data from a single table. In chapter 4, you'll learn how to get data from two or more tables. In chapter 5, you'll learn how to summarize the data that you retrieve. And in chapter 6, you'll learn how to code subqueries, which are SELECT statements coded within other statements.

Next, chapter 7 shows you how to use the INSERT, UPDATE, and DELETE statements to add, update, and delete rows in a table. Finally, chapter 8 shows you how to work with the various types of data that SQL Server supports and how to use some of the SQL Server functions for working with data in your SQL statements. When you complete these chapters, you'll have the skills you need to code most any SELECT, INSERT, UPDATE, or DELETE statement.

3

How to retrieve data from a single table

In this chapter, you'll learn how to code SELECT statements that retrieve data from a single table. You should realize, though, that the skills covered here are the essential ones that apply to any SELECT statement you code…no matter how many tables it operates on, no matter how complex the retrieval. So you'll want to be sure you have a good understanding of the material in this chapter before you go on to the chapters that follow.

An introduction to the SELECT statement

To help you learn to code SELECT statements, this chapter starts by presenting its basic syntax. Next, it presents several examples that will give you an idea of what you can do with this statement. Then, the rest of this chapter will teach you the details of coding this statement.

The basic syntax of the SELECT statement

Figure 3-1 presents the basic syntax of the SELECT statement. The syntax summary at the top of this figure uses conventions that are similar to those used in other programming manuals. Capitalized words are *keywords* that you have to type exactly as shown. In contrast, you have to provide replacements for the lowercase words. For example, you can enter a list of columns in place of *select_list*, and you can enter a table name in place of *table_source*.

Beyond that, you can choose between the items in a syntax summary that are separated by pipes (|) and enclosed in braces ({}) or brackets ([]). And you can omit items enclosed in brackets. If you have a choice between two or more optional items, the default item is underlined. And if an element can be coded multiple times in a statement, it's followed by an ellipsis (…). You'll see examples of pipes, braces, default values, and ellipses in syntax summaries later in this chapter. For now, if you compare the syntax in this figure with the coding examples in the next figure, you should easily see how the two are related.

The syntax summary in this figure has been simplified so that you can focus on the four main clauses of the SELECT statement: SELECT, FROM, WHERE, and ORDER BY. Most of the SELECT statements you code will contain all four of these clauses. However, only the SELECT and FROM clauses are required.

The SELECT clause is always the first clause in a SELECT statement. It identifies the columns that will be included in the result set. These columns are retrieved from the base tables named in the FROM clause. Since this chapter focuses on retrieving data from a single table, the FROM clauses in all of the statements shown in this chapter name a single base table. In the next chapter, though, you'll learn how to retrieve data from two or more tables.

The WHERE and ORDER BY clauses are optional. The ORDER BY clause determines how the rows in the result set are sorted, and the WHERE clause determines which rows in the base table are included in the result set. The WHERE clause specifies a search condition that's used to *filter* the rows in the base table. This search condition can consist of one or more *Boolean expressions*, or *predicates*. A Boolean expression is an expression that evaluates to True or False. When the search condition evaluates to True, the row is included in the result set.

In this book, I won't use the terms "Boolean expression" or "predicate" because I don't think they clearly describe the content of the WHERE clause. Instead, I'll just use the term "search condition" to refer to an expression that evaluates to True or False.

The simplified syntax of the SELECT statement

```
SELECT select_list
FROM table_source
[WHERE search_condition]
[ORDER BY order_by_list]
```

The four clauses of the SELECT statement

Clause	Description
SELECT	Describes the columns that will be included in the result set.
FROM	Names the table from which the query will retrieve the data.
WHERE	Specifies the conditions that must be met for a row to be included in the result set. This clause is optional.
ORDER BY	Specifies how the rows in the result set will be sorted. This clause is optional.

Description

- You use the basic SELECT statement shown above to retrieve the columns specified in the SELECT clause from the base table specified in the FROM clause and store them in a result set.
- The WHERE clause is used to *filter* the rows in the base table so that only those rows that match the search condition are included in the result set. If you omit the WHERE clause, all of the rows in the base table are included.
- The search condition of a WHERE clause consists of one or more *Boolean expressions*, or *predicates*, that result in a value of True, False, or Unknown. If the combination of all the expressions is True, the row being tested is included in the result set. Otherwise, it's not.
- If you include the ORDER BY clause, the rows in the result set are sorted in the specified sequence. Otherwise, the rows are returned in the same order as they appear in the base table. In most cases, that means that they're returned in primary key sequence.

Note

- The syntax shown above does not include all of the clauses of the SELECT statement. You'll learn about the other clauses later in this book.

Figure 3-1 The basic syntax of the SELECT statement

SELECT statement examples

Figure 3-2 presents five SELECT statement examples. All of these statements retrieve data from the Invoices table. If you aren't already familiar with this table, you should use the Management Studio as described in the last chapter to review its definition.

The first statement in this figure retrieves all of the rows and columns from the Invoices table. Here, an asterisk (*) is used as a shorthand to indicate that all of the columns should be retrieved, and the WHERE clause is omitted so that there are no conditions on the rows that are retrieved. Notice that this statement doesn't include an ORDER BY clause, so the rows are in primary key sequence. You can see the results following this statement as they're displayed by the Management Studio. Notice that both horizontal and vertical scroll bars are displayed, indicating that the result set contains more rows and columns than can be displayed on the screen at one time.

The second statement retrieves selected columns from the Invoices table. As you can see, the columns to be retrieved are listed in the SELECT clause. Like the first statement, this statement doesn't include a WHERE clause, so all the rows are retrieved. Then, the ORDER BY clause causes the rows to be sorted by the InvoiceTotal column in ascending sequence.

The third statement also lists the columns to be retrieved. In this case, though, the last column is calculated from two columns in the base table, CreditTotal and PaymentTotal, and the resulting column is given the name TotalCredits. In addition, the WHERE clause specifies that only the invoice whose InvoiceID column has a value of 17 should be retrieved.

The fourth SELECT statement includes a WHERE clause whose condition specifies a range of values. In this case, only invoices with invoice dates between 05/01/2008 and 05/31/2008 are retrieved. In addition, the rows in the result set are sorted by invoice date.

The last statement in this figure shows another variation of the WHERE clause. In this case, only those rows with invoice totals greater than 50,000 are retrieved. Since none of the rows in the Invoices table satisfy this condition, the result set is empty.

A SELECT statement that retrieves all the data from the Invoices table

```
SELECT *
FROM Invoices
```

	InvoiceID	VendorID	InvoiceNumber	InvoiceDate	InvoiceTotal	PaymentTotal	CreditTotal
1	1	122	989319-457	2008-04-08 00:00:00	3813.33	3813.33	0.00
2	2	123	263253241	2008-04-10 00:00:00	40.20	40.20	0.00
3	3	123	963253234	2008-04-13 00:00:00	138.75	138.75	0.00
4	4	123	2-000-2993	2008-04-16 00:00:00	144.70	144.70	0.00

```
(114 rows)
```

A SELECT statement that retrieves three columns from each row, sorted in ascending sequence by invoice total

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
ORDER BY InvoiceTotal
```

	InvoiceNumber	InvoiceDate	InvoiceTotal
1	25022117	2008-05-01 00:00:00	6.00
2	24863706	2008-05-10 00:00:00	6.00
3	24780512	2008-06-22 00:00:00	6.00
4	21-4923721	2008-05-13 00:00:00	9.95

```
(114 rows)
```

A SELECT statement that retrieves two columns and a calculated value for a specific invoice

```
SELECT InvoiceID, InvoiceTotal, CreditTotal + PaymentTotal AS TotalCredits
FROM Invoices
WHERE InvoiceID = 17
```

	InvoiceID	InvoiceTotal	TotalCredits
1	17	10.00	10.00

A SELECT statement that retrieves all invoices between given dates

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE InvoiceDate BETWEEN '2008-05-01' AND '2008-05-31'
ORDER BY InvoiceDate
```

	InvoiceNumber	InvoiceDate	InvoiceTotal
1	25022117	2008-05-01 00:00:00	6.00
2	21-4748363	2008-05-03 00:00:00	9.95
3	P02-88D77S7	2008-05-03 00:00:00	856.92
4	4-321-2596	2008-05-05 00:00:00	10.00

```
(29 rows)
```

A SELECT statement that returns an empty result set

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal
FROM Invoices
WHERE InvoiceTotal > 50000
```

	InvoiceNumber	InvoiceDate	InvoiceTotal

Figure 3-2 SELECT statement examples

How to code the SELECT clause

Figure 3-3 presents an expanded syntax for the SELECT clause. The keywords shown in the first line allow you to restrict the rows that are returned by a query. You'll learn how to code them in a few minutes. First, though, you'll learn various techniques for identifying which columns are to be included in a result set.

How to code column specifications

Figure 3-3 summarizes the techniques you can use to code column specifications. You saw how to use some of these techniques in the previous figure. For example, you can code an asterisk in the SELECT clause to retrieve all of the columns in the base table, and you can code a list of column names separated by commas. Note that when you code an asterisk, the columns are returned in the order that they occur in the base table.

You can also code a column specification as an *expression*. For example, you can use an arithmetic expression to perform a calculation on two or more columns in the base table, and you can use a string expression to combine two or more string values. An expression can also include one or more functions. You'll learn more about each of these techniques in the topics that follow.

But first, you should know that when you code the SELECT clause, you should include only the columns you need. For example, you shouldn't code an asterisk to retrieve all the columns unless you need all the columns. That's because the amount of data that's retrieved can affect system performance. This is particularly important if you're developing SQL statements that will be used by application programs.

The expanded syntax of the SELECT clause

```
SELECT [ALL|DISTINCT] [TOP n [PERCENT] [WITH TIES]]
    column_specification [[AS] result_column]
    [, column_specification [[AS] result_column]] ...
```

Five ways to code column specifications

Source	Option	Syntax
Base table value	All columns	*
	Column name	column_name
Calculated value	Result of a calculation	Arithmetic expression (see figure 3-6)
	Result of a concatenation	String expression (see figure 3-5)
	Result of a function	Function (see figure 3-7)

Column specifications that use base table values

The * is used to retrieve all columns

```
SELECT *
```

Column names are used to retrieve specific columns

```
SELECT VendorName, VendorCity, VendorState
```

Column specifications that use calculated values

An arithmetic expression is used to calculate BalanceDue

```
SELECT InvoiceNumber,
    InvoiceTotal - PaymentTotal – CreditTotal AS BalanceDue
```

A string expression is used to calculate FullName

```
SELECT VendorContactFName + ' ' + VendorContactLName AS FullName
```

A function is used to calculate CurrentDate

```
SELECT InvoiceNumber, InvoiceDate,
    GETDATE() AS CurrentDate
```

Description

- Use SELECT * only when you need to retrieve all of the columns from a table. Otherwise, list the names of the columns you need.
- An *expression* is a combination of column names and operators that evaluate to a single value. In the SELECT clause, you can code arithmetic expressions, string expressions, and expressions that include one or more functions.
- After each column specification, you can code an AS clause to specify the name for the column in the result set. See figure 3-4 for details.

Note

- The other elements shown in the syntax summary above let you control the number of rows that are returned by a query. You can use the ALL and DISTINCT keywords to determine whether or not duplicate rows are returned. And you can use the TOP clause to retrieve a specific number or percent of rows. See figures 3-8 and 3-9 for details.

Figure 3-3 How to code column specifications

How to name the columns in a result set

By default, a column in a result set is given the same name as the column in the base table. However, you can specify a different name if you need to. You can also name a column that contains a calculated value. When you do that, the new column name is called a *column alias*. Figure 3-4 presents two techniques for creating column aliases.

The first technique is to code the column specification followed by the AS keyword and the column alias. This is the ANSI-standard coding technique, and it's illustrated by the first example in this figure. Here, a space is added between the two words in the name of the InvoiceNumber column, the InvoiceDate column is changed to just Date, and the InvoiceTotal column is changed to Total. Notice that because a space is included in the name of the first column, it's enclosed in brackets ([]). As you'll learn in chapter 10, any name that doesn't follow SQL Server's rules for naming objects must be enclosed in either brackets or double quotes. Column aliases can also be enclosed in single quotes.

The second example in this figure illustrates another technique for creating a column alias. Here, the column is assigned to an alias using an equal sign. This technique is only available with SQL Server, not with other types of databases, and is included for compatibility with earlier versions of SQL Server. So although you may see this technique used in older code, I don't recommend it for new statements you write.

The third example in this figure illustrates what happens when you don't assign an alias to a calculated column. Here, no name is assigned to the column, which usually isn't what you want. That's why you usually assign a name to any column that's calculated from other columns in the base table.

Two SELECT statements that name the columns in the result set

A SELECT statement that uses the AS keyword (the preferred technique)

```
SELECT InvoiceNumber AS [Invoice Number], InvoiceDate AS Date,
    InvoiceTotal AS Total
FROM Invoices
```

A SELECT statement that uses the equal operator (an older technique)

```
SELECT [Invoice Number] = InvoiceNumber, Date = InvoiceDate,
    Total = InvoiceTotal
FROM Invoices
```

The result set for both SELECT statements

	Invoice Number	Date	Total
1	989319-457	2008-04-08 00:00:00	3813.33
2	263253241	2008-04-10 00:00:00	40.20
3	963253234	2008-04-13 00:00:00	138.75
4	2-000-2993	2008-04-16 00:00:00	144.70
5	963253251	2008-04-16 00:00:00	15.50

A SELECT statement that doesn't provide a name for a calculated column

```
SELECT InvoiceNumber, InvoiceDate, InvoiceTotal,
    InvoiceTotal - PaymentTotal - CreditTotal
FROM Invoices
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	(No column name)
1	989319-457	2008-04-08 00:00:00	3813.33	0.00
2	263253241	2008-04-10 00:00:00	40.20	0.00
3	963253234	2008-04-13 00:00:00	138.75	0.00
4	2-000-2993	2008-04-16 00:00:00	144.70	0.00
5	963253251	2008-04-16 00:00:00	15.50	0.00

Description

- By default, a column in the result set is given the same name as the column in the base table. If that's not what you want, you can specify a *column alias* or *substitute name* for the column.
- One way to name a column is to use the AS phrase as shown in the first example above. Although the AS keyword is optional, I recommend you code it for readability.
- Another way to name a column is to code the name followed by an equal sign and the column specification as shown in the second example above. This syntax is unique to Transact-SQL.
- It's generally considered a good practice to specify an alias for a column that contains a calculated value. If you don't, no name is assigned to it as shown in the third example above.
- If an alias includes spaces or special characters, you must enclose it in double quotes or brackets ([]). That's true of all names you use in Transact-SQL. SQL Server also lets you enclose column aliases in single quotes for compatibility with earlier releases.

Figure 3-4 How to name the columns in a result set

How to code string expressions

A *string expression* consists of a combination of one or more character columns and *literal values*. To combine, or *concatenate*, the columns and values, you use the *concatenation operator* (+). This is illustrated by the examples in figure 3-5.

The first example shows how to concatenate the VendorCity and VendorState columns in the Vendors table. Notice that because no alias is assigned to this column, it doesn't have a name in the result set. Also notice that the data in the VendorState column appears immediately after the data in the VendorCity column in the results. That's because of the way VendorCity is defined in the database. Because it's defined as a variable-length column (the varchar data type), only the actual data in the column is included in the result. In contrast, if the column had been defined with a fixed length, any spaces following the name would have been included in the result. You'll learn about data types and how they affect the data in your result set in chapter 8.

The second example shows how to format a string expression by adding spaces and punctuation. Here, the VendorCity column is concatenated with a *string literal*, or *string constant*, that contains a comma and a space. Then, the VendorState column is concatenated with that result, followed by a string literal that contains a single space and the VendorZipCode column.

Occasionally, you may need to include a single quotation mark or an apostrophe within a literal string. If you simply type a single quote, however, the system will misinterpret it as the end of the literal string. As a result, you must code two quotation marks in a row. This is illustrated by the third example in this figure.

How to concatenate string data

```
SELECT VendorCity, VendorState, VendorCity + VendorState
FROM Vendors
```

	VendorCity	VendorState	(No column name)
1	Madison	WI	MadisonWI
2	Washington	DC	WashingtonDC
3	Washington	DC	WashingtonDC

How to format string data using literal values

```
SELECT VendorName,
    VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
```

	VendorName	Address
1	US Postal Service	Madison, WI 53707
2	National Information Data Ctr	Washington, DC 20090
3	Register of Copyrights	Washington, DC 20559
4	Jobtrak	Los Angeles, CA 90025

How to include apostrophes in literal values

```
SELECT VendorName + '''s Address: ',
    VendorCity + ', ' + VendorState + ' ' + VendorZipCode
FROM Vendors
```

	(No column name)	(No column name)
1	US Postal Service's Address:	Madison, WI 53707
2	National Information Data Ctr's Address:	Washington, DC 20090
3	Register of Copyrights's Address:	Washington, DC 20559
4	Jobtrak's Address:	Los Angeles, CA 90025
5	Newbrige Book Clubs's Address:	Washington, NJ 07882
6	California Chamber Of Commerce's Address:	Sacramento, CA 95827

Description

- A *string expression* can consist of one or more character columns, one or more *literal values*, or a combination of character columns and literal values.
- The columns specified in a string expression must contain string data (that means they're defined with the char or varchar data type).
- The literal values in a string expression also contain string data, so they can be called *string literals* or *string constants*. To create a literal value, enclose one or more characters within single quotation marks (').
- You can use the *concatenation operator* (+) to combine columns and literals in a string expression.
- You can include a single quote within a literal value by coding two single quotation marks as shown in the third example above.

Figure 3-5 How to code string expressions

How to code arithmetic expressions

Figure 3-6 shows how to code *arithmetic expressions*. To start, it summarizes the five *arithmetic operators* you can use in this type of expression. Then, it presents three examples that illustrate how you use these operators.

The SELECT statement in the first example includes an arithmetic expression that calculates the balance due for an invoice. This expression subtracts the PaymentTotal and CreditTotal columns from the InvoiceTotal column. The resulting column is given the name BalanceDue.

When SQL Server evaluates an arithmetic expression, it performs the operations from left to right based on the *order of precedence*. This order says that multiplication, division, and modulo operations are done first, followed by addition and subtraction. If that's not what you want, you can use parentheses to specify how you want an expression evaluated. Then, the expressions in the innermost sets of parentheses are evaluated first, followed by the expressions in outer sets of parentheses. Within each set of parentheses, the expression is evaluated from left to right in the order of precedence. Of course, you can also use parentheses to clarify an expression even if they're not needed for the expression to be evaluated properly.

To illustrate how parentheses and the order of precedence affect the evaluation of an expression, consider the second example in this figure. Here, the expressions in the second and third columns both use the same operators. When SQL Server evaluates the expression in the second column, it performs the multiplication operation before the addition operation because multiplication comes before addition in the order of precedence. When SQL Server evaluates the expression in the third column, however, it performs the addition operation first because it's enclosed in parentheses. As you can see in the result set shown here, these two expressions result in different values.

Although you're probably familiar with the addition, subtraction, multiplication, and division operators, you may not be familiar with the modulo operator. This operator returns the remainder of a division of two integers. This is illustrated in the third example in this figure. Here, the second column contains an expression that returns the quotient of a division operation. Note that the result of the division of two integers is always an integer. You'll learn more about that in chapter 8. The third column contains an expression that returns the remainder of the division operation. If you study this example for a minute, you should quickly see how this works.

The arithmetic operators in order of precedence

*	Multiplication
/	Division
%	Modulo (Remainder)
+	Addition
-	Subtraction

A SELECT statement that calculates the balance due

```
SELECT InvoiceTotal, PaymentTotal, CreditTotal,
    InvoiceTotal - PaymentTotal - CreditTotal AS BalanceDue
FROM Invoices
```

	InvoiceTotal	PaymentTotal	CreditTotal	BalanceDue
1	3813.33	3813.33	0.00	0.00
2	40.20	40.20	0.00	0.00
3	138.75	138.75	0.00	0.00

A SELECT statement that uses parentheses to control the sequence of operations

```
SELECT InvoiceID,
    InvoiceID + 7 * 3 AS OrderOfPrecedence,
    (InvoiceID + 7) * 3 AS AddFirst
FROM Invoices
ORDER BY InvoiceID
```

	InvoiceID	OrderOfPrecedence	AddFirst
1	1	22	24
2	2	23	27
3	3	24	30

A SELECT statement that uses the modulo operator

```
SELECT InvoiceID,
    InvoiceID / 10 AS Quotient,
    InvoiceID % 10 AS Remainder
FROM Invoices
ORDER BY InvoiceID
```

	InvoiceID	Quotient	Remainder
9	9	0	9
10	10	1	0
11	11	1	1

Description

- Unless parentheses are used, the operations in an expression take place from left to right in the *order of precedence*. For arithmetic expressions, multiplication, division, and modulo operations are done first, followed by addition and subtraction.
- Whenever necessary, you can use parentheses to clarify or override the sequence of operations. Then, the operations in the innermost sets of parentheses are done first, followed by the operations in the next sets, and so on.

Figure 3-6 How to code arithmetic expressions

How to use functions

Figure 3-7 introduces you to *functions* and illustrates how you use them in column specifications. A function performs an operation and returns a value. For now, don't worry about the details of how the functions shown here work. You'll learn more about all of these functions in chapter 8. Instead, just focus on how they're used in column specifications.

To code a function, you begin by entering its name followed by a set of parentheses. If the function requires one or more *parameters*, you enter them within the parentheses and separate them with commas. When you enter a parameter, you need to be sure it has the correct data type. You'll learn more about that in chapter 8.

The first example in this figure shows how to use the LEFT function to extract the first character of the VendorContactFName and VendorContactLName columns. The first parameter of this function specifies the string values, and the second parameter specifies the number of characters to return. The results of the two functions are then concatenated to form initials as shown in the result set for this statement.

The second example shows how to use the CONVERT function to change the data type of a value. This function requires two parameters. The first parameter specifies the new data type, and the second parameter specifies the value to convert. In addition, this function accepts an optional third parameter that specifies the format of the returned value. The first CONVERT function shown here, for example, converts the PaymentDate column to a character value with the format mm/dd/yy. And the second CONVERT function converts the PaymentTotal column to a variable-length character value that's formatted with commas. These functions are included in a string expression that concatenates their return values with the InvoiceNumber column and three literal values.

The third example uses two functions that work with dates. The first one, GETDATE, returns the current date. Notice that although this function doesn't accept any parameters, the parentheses are still included. The second function, DATEDIFF, gets the difference between two date values. This function requires three parameters. The first one specifies the units in which the result will be expressed. In this example, the function will return the number of days between the two dates. The second and third parameters specify the start date and the end date. Here, the second parameter is the invoice date and the third parameter is the current date, which is obtained using the GETDATE function.

A SELECT statement that uses the LEFT function

```
SELECT VendorContactFName, VendorContactLName,
    LEFT(VendorContactFName, 1) +
    LEFT(VendorContactLName, 1) AS Initials
FROM Vendors
```

	VendorContactFName	VendorContactLName	Initials
1	Francesco	Alberto	FA
2	Ania	Irvin	AI
3	Lukas	Liana	LL

A SELECT statement that uses the CONVERT function

```
SELECT 'Invoice: #' + InvoiceNumber
    + ', dated ' + CONVERT(char(8), PaymentDate, 1)
    + ' for $' + CONVERT(varchar(9), PaymentTotal, 1)
FROM Invoices
```

	(No column name)
1	Invoice: #989319-457, dated 05/07/08 for $3,813.33
2	Invoice: #263253241, dated 05/14/08 for $40.20

A SELECT statement that computes the age of an invoice

```
SELECT InvoiceDate,
    GETDATE() AS 'Today''s Date',
    DATEDIFF(day, InvoiceDate, GETDATE()) AS Age
FROM Invoices
```

	InvoiceDate	Today's Date	Age
1	2008-08-02 00:00:00	2008-10-02 15:16:29.373	61
2	2008-08-01 00:00:00	2008-10-02 15:16:29.373	62
3	2008-07-31 00:00:00	2008-10-02 15:16:29.373	63

Description

- An expression can include any of the *functions* that are supported by SQL Server. A function performs an operation and returns a value.
- A function consists of the function name, followed by a set of parentheses that contains any *parameters*, or *arguments*, required by the function. If a function requires two or more arguments, you separate them with commas.
- For more information on using functions, see chapter 8.

Figure 3-7 How to use functions

How to use the DISTINCT keyword to eliminate duplicate rows

By default, all of the rows in the base table that satisfy the search condition you specify in the WHERE clause are included in the result set. In some cases, though, that means that the result set will contain duplicate rows, or rows whose column values are identical. If that's not what you want, you can include the DISTINCT keyword in the SELECT clause to eliminate the duplicate rows.

Figure 3-8 illustrates how this works. Here, both SELECT statements retrieve the VendorCity and VendorState columns from the Vendors table. The first statement, however, doesn't include the DISTINCT keyword. Because of that, the same city and state can appear in the result set multiple times. In the results shown in this figure, for example, you can see that Anaheim CA occurs twice and Boston MA occurs three times. In contrast, the second statement includes the DISTINCT keyword, so each city/state combination is included only once.

A SELECT statement that returns all rows

```
SELECT VendorCity, VendorState
FROM Vendors
ORDER BY VendorCity
```

	VendorCity	VendorState
1	Anaheim	CA
2	Anaheim	CA
3	Ann Arbor	MI
4	Auburn Hills	MI
5	Boston	MA
6	Boston	MA
7	Boston	MA
8	Brea	CA

```
(122 rows)
```

A SELECT statement that eliminates duplicate rows

```
SELECT DISTINCT VendorCity, VendorState
FROM Vendors
```

	VendorCity	VendorState
1	Anaheim	CA
2	Ann Arbor	MI
3	Auburn Hills	MI
4	Boston	MA
5	Brea	CA
6	Carol Stream	IL
7	Charlotte	NC
8	Chicago	IL

```
(53 rows)
```

Description

- The DISTINCT keyword prevents duplicate (identical) rows from being included in the result set. It also causes the result set to be sorted by its first column.
- The ALL keyword causes all rows matching the search condition to be included in the result set, regardless of whether rows are duplicated. Since this is the default, it's a common practice to omit the ALL keyword.
- To use the DISTINCT or ALL keyword, code it immediately after the SELECT keyword as shown above.

Figure 3-8 How to use the DISTINCT keyword to eliminate duplicate rows

How to use the TOP clause to return a subset of selected rows

In addition to eliminating duplicate rows, you can limit the number of rows that are retrieved by a SELECT statement. To do that, you use the TOP clause. Figure 3-9 shows you how.

You can use the TOP clause in one of two ways. First, you can use it to retrieve a specific number of rows from the beginning, or top, of the result set. To do that, you code the TOP keyword followed by an integer value that specifies the number of rows to be returned. This is illustrated in the first example in this figure. Here, only five rows are returned. Notice that this statement also includes an ORDER BY clause that sorts the rows by the InvoiceTotal column in descending sequence. That way, the invoices with the highest invoice totals will be returned.

You can also use the TOP clause to retrieve a specific percent of the rows in the result set. To do that, you include the PERCENT keyword as shown in the second example. In this case, the result set includes six rows, which is five percent of the total of 122 rows.

By default, the TOP clause causes the exact number or percent of rows you specify to be retrieved. However, if additional rows match the values in the last row, you can include those additional rows by including WITH TIES in the TOP clause. This is illustrated in the third example in this figure. Here, the SELECT statement says to retrieve the top five rows from a result set that includes the VendorID and InvoiceDate columns sorted by the InvoiceDate column. As you can see, however, the result set includes six rows instead of five. That's because WITH TIES is included in the TOP clause, and the columns in the sixth row have the same values as the columns in the fifth row.

A SELECT statement with a TOP clause

```
SELECT TOP 5 VendorID, InvoiceTotal
FROM Invoices
ORDER BY InvoiceTotal DESC
```

	VendorID	InvoiceTotal
1	110	37966.19
2	110	26881.40
3	110	23517.58
4	72	21842.00
5	110	20551.18

A SELECT statement with a TOP clause and the PERCENT keyword

```
SELECT TOP 5 PERCENT VendorID, InvoiceTotal
FROM Invoices
ORDER BY InvoiceTotal DESC
```

	VendorID	InvoiceTotal
1	110	37966.19
2	110	26881.40
3	110	23517.58
4	72	21842.00
5	110	20551.18
6	110	10976.06

A SELECT statement with a TOP clause and the WITH TIES keyword

```
SELECT TOP 5 WITH TIES VendorID, InvoiceDate
FROM Invoices
ORDER BY InvoiceDate DESC
```

	VendorID	InvoiceDate
1	122	2008-04-08 00:00:00
2	123	2008-04-10 00:00:00
3	123	2008-04-13 00:00:00
4	123	2008-04-16 00:00:00
5	123	2008-04-16 00:00:00
6	123	2008-04-16 00:00:00

Description

- You can use the TOP clause within a SELECT clause to limit the number of rows included in the result set. When you use this clause, the first *n* rows that meet the search condition are included, where *n* is an integer.
- If you include PERCENT, the first *n* percent of the selected rows are included in the result set.
- If you include WITH TIES, additional rows will be included if their values match, or *tie,* the values of the last row.
- You should include an ORDER BY clause whenever you use the TOP keyword. Otherwise, the rows in the result set will be in no particular sequence.

Figure 3-9 How to use the TOP clause to return a subset of selected rows

How to code the WHERE clause

Earlier in this chapter, I mentioned that to improve performance, you should code your SELECT statements so they retrieve only the columns you need. That goes for retrieving rows too: The fewer rows you retrieve, the more efficient the statement will be. Because of that, you'll almost always include a WHERE clause on your SELECT statements with a search condition that filters the rows in the base table so that only the rows you need are retrieved. In the topics that follow, you'll learn a variety of ways to code this clause.

How to use comparison operators

Figure 3-10 shows you how to use the *comparison operators* in the search condition of a WHERE clause. As you can see in the syntax summary at the top of this figure, you use a comparison operator to compare two expressions. If the result of the comparison is True, the row being tested is included in the query results.

The examples in this figure show how to use some of the comparison operators. The first WHERE clause, for example, uses the equal operator (=) to retrieve only those rows whose VendorState column have a value of IA. Since the state code is a string literal, it must be included in single quotes. In contrast, the numeric literal used in the second WHERE clause is not enclosed in quotes. This clause uses the greater than (>) operator to retrieve only those rows that have a balance due greater than zero.

The third WHERE clause illustrates another way to retrieve all the invoices with a balance due. Like the second clause, it uses the greater than operator. Instead of comparing the balance due to a value of zero, however, it compares the invoice total to the total of the payments and credits that have been applied to the invoice.

The fourth WHERE clause illustrates how you can use comparison operators other than the equal operator with string data. In this example, the less than operator (<) is used to compare the value of the VendorName column to a literal string that contains the letter M. That will cause the query to return all vendors with names that begin with the letters A through L.

You can also use the comparison operators with date literals, as illustrated by the fifth and sixth WHERE clauses. The fifth clause will retrieve rows with invoice dates on or before May 31, 2008, and the sixth clause will retrieve rows with invoice dates on or after May 1, 2008. Like string literals, date literals must be enclosed in single quotes. In addition, you can use different formats to specify dates as shown by the two date literals shown in this figure. You'll learn more about the acceptable date formats in chapter 8.

The last WHERE clause shows how you can test for a not equal condition. To do that, you code a less than sign followed by a greater than sign. In this case, only rows with a credit total that's not equal to zero will be retrieved.

The syntax of the WHERE clause with comparison operators

```
WHERE expression_1 operator expression_2
```

The comparison operators

=	Equal
>	Greater than
<	Less than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal

Examples of WHERE clauses that retrieve…

Vendors located in Iowa

```
WHERE VendorState = 'IA'
```

Invoices with a balance due (two variations)

```
WHERE InvoiceTotal - PaymentTotal - CreditTotal > 0
WHERE InvoiceTotal > PaymentTotal + CreditTotal
```

Vendors with names from A to L

```
WHERE VendorName < 'M'
```

Invoices on or before a specified date

```
WHERE InvoiceDate <= '2008-05-31'
```

Invoices on or after a specified date

```
WHERE InvoiceDate >= '5/1/08'
```

Invoices with credits that don't equal zero

```
WHERE CreditTotal <> 0
```

Description

- You can use a comparison operator to compare any two expressions that result in like data types. Although unlike data types may be converted to data types that can be compared, the comparison may produce unexpected results.
- If a comparison results in a True value, the row being tested is included in the result set. If it's False or Unknown, the row isn't included.
- To use a string literal or a *date literal* in a comparison, enclose it in quotes. To use a numeric literal, enter the number without quotes.
- Character comparisons performed on SQL Server databases are not case-sensitive. So, for example, 'CA' and 'Ca' are considered equivalent.

Figure 3-10 How to use the comparison operators

Whenever possible, you should compare expressions that have similar data types. If you attempt to compare expressions that have different data types, SQL Server may implicitly convert the data type for you. Often, this implicit conversion is acceptable. However, implicit conversions will occasionally yield unexpected results. In that case, you can use the CONVERT function you saw earlier in this chapter or the CAST function you'll learn about in chapter 8 to explicitly convert data types so the comparison yields the results you want.

How to use the AND, OR, and NOT logical operators

Figure 3-11 shows how to use *logical operators* in a WHERE clause. You can use the AND and OR operators to combine two or more search conditions into a *compound condition*. And you can use the NOT operator to negate a search condition. The examples in this figure illustrate how these operators work.

The first two examples illustrate the difference between the AND and OR operators. When you use the AND operator, both conditions must be true. So, in the first example, only those vendors in New Jersey whose year-to-date purchases are greater than 200 are retrieved from the Vendors table. When you use the OR operator, though, only one of the conditions must be true. So, in the second example, all the vendors from New Jersey and all the vendors whose year-to-date purchases are greater than 200 are retrieved.

The third example shows a compound condition that uses two NOT operators. As you can see, this expression is somewhat difficult to understand. Because of that, and because using the NOT operator can reduce system performance, you should avoid using this operator whenever possible. The fourth example in this figure, for instance, shows how the search condition in the third example can be rephrased to eliminate the NOT operator. Notice that the condition in the fourth example is much easier to understand.

The last two examples in this figure show how the order of precedence for the logical operators and the use of parentheses affect the result of a search condition. By default, the NOT operator is evaluated first, followed by AND and then OR. However, you can use parentheses to override the order of precedence or to clarify a logical expression, just as you can with arithmetic expressions. In the next to last example, for instance, no parentheses are used, so the two conditions connected by the AND operator are evaluated first. In the last example, though, parentheses are used so that the two conditions connected by the OR operator are evaluated first. If you take a minute to review the results shown in this figure, you should be able to see how these two conditions differ.

The syntax of the WHERE clause with logical operators

```
WHERE [NOT] search_condition_1 {AND|OR} [NOT] search_condition_2 ...
```

Examples of queries using logical operators

A search condition that uses the AND operator

```
WHERE VendorState = 'NJ' AND YTDPurchases > 200
```

A search condition that uses the OR operator

```
WHERE VendorState = 'NJ' OR YTDPurchases > 200
```

A search condition that uses the NOT operator

```
WHERE NOT (InvoiceTotal >= 5000 OR NOT InvoiceDate <= '2008-07-01')
```

The same condition rephrased to eliminate the NOT operator

```
WHERE InvoiceTotal < 5000 AND InvoiceDate <= '2008-07-01'
```

A compound condition without parentheses

```
WHERE InvoiceDate > '05/01/2008'
    OR InvoiceTotal > 500
    AND InvoiceTotal - PaymentTotal - CreditTotal > 0
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	BalanceDue
1	P02-88D77S7	2008-05-03 00:00:00	856.92	0.00
2	21-4748363	2008-05-03 00:00:00	9.95	0.00
3	4-321-2596	2008-05-05 00:00:00	10.00	0.00
4	963253242	2008-05-06 00:00:00	104.00	0.00

```
(100 rows)
```

The same compound condition with parentheses

```
WHERE (InvoiceDate > '05/01/2008'
    OR InvoiceTotal > 500)
    AND InvoiceTotal - PaymentTotal - CreditTotal > 0
```

	InvoiceNumber	InvoiceDate	InvoiceTotal	BalanceDue
1	39104	2008-07-10 00:00:00	85.31	85.31
2	963253264	2008-07-18 00:00:00	52.25	52.25
3	31361833	2008-07-21 00:00:00	579.42	579.42
4	263253268	2008-07-21 00:00:00	59.97	59.97

```
(11 rows)
```

Description

- You can use the AND and OR *logical operators* to create *compound conditions* that consist of two or more conditions. You use the AND operator to specify that the search must satisfy both of the conditions, and you use the OR operator to specify that the search must satisfy at least one of the conditions.
- You can use the NOT operator to negate a condition. Because this operator can make the search condition difficult to read, you should rephrase the condition if possible so it doesn't use NOT.
- When SQL Server evaluates a compound condition, it evaluates the operators in this sequence: (1) NOT, (2) AND, and (3) OR. You can use parentheses to override this order of precedence or to clarify the sequence in which the operations will be evaluated.

Figure 3-11 How to use the AND, OR, and NOT logical operators

How to use the IN operator

Figure 3-12 shows how to code a WHERE clause that uses the IN operator. When you use this operator, the value of the test expression is compared with the list of expressions in the IN phrase. If the test expression is equal to one of the expressions in the list, the row is included in the query results. This is illustrated by the first example in this figure, which will return all rows whose TermsID column is equal to 1, 3, or 4.

You can also use the NOT operator with the IN phrase to test for a value that's not in a list of expressions. This is illustrated by the second example in this figure. In this case, only those vendors who are not in California, Nevada, or Oregon are retrieved.

If you look at the syntax of the IN phrase shown at the top of this figure, you'll see that you can code a *subquery* in place of a list of expressions. Subqueries are a powerful tool that you'll learn about in detail in chapter 6. For now, though, you should know that a subquery is simply a SELECT statement within another statement. In the third example in this figure, for instance, a subquery is used to return a list of VendorID values for vendors who have invoices dated May 1, 2008. Then, the WHERE clause retrieves a vendor row only if the vendor is in that list. Note that for this to work, the subquery must return a single column, in this case, VendorID.

The syntax of the WHERE clause with an IN phrase

```
WHERE test_expression [NOT] IN ({subquery|expression_1 [, expression_2]...})
```

Examples of the IN phrase

An IN phrase with a list of numeric literals

```
WHERE TermsID IN (1, 3, 4)
```

An IN phrase preceded by NOT

```
WHERE VendorState NOT IN ('CA', 'NV', 'OR')
```

An IN phrase with a subquery

```
WHERE VendorID IN
    (SELECT VendorID
     FROM Invoices
     WHERE InvoiceDate = '2008-05-01')
```

Description

- You can use the IN phrase to test whether an expression is equal to a value in a list of expressions. Each of the expressions in the list must evaluate to the same type of data as the test expression.
- The list of expressions can be coded in any order without affecting the order of the rows in the result set.
- You can use the NOT operator to test for an expression that's not in the list of expressions.
- You can also compare the test expression to the items in a list returned by a *subquery* as illustrated by the third example above. You'll learn more about coding subqueries in chapter 6.

Figure 3-12 How to use the IN operator

How to use the BETWEEN operator

Figure 3-13 shows how to use the BETWEEN operator in a WHERE clause. When you use this operator, the value of a test expression is compared to the range of values specified in the BETWEEN phrase. If the value falls within this range, the row is included in the query results.

The first example in this figure shows a simple WHERE clause that uses the BETWEEN operator. It retrieves invoices with invoice dates between May 1, 2008 and May 31, 2008. Note that the range is inclusive, so invoices with invoice dates of May 1 and May 31 are included in the results.

The second example shows how to use the NOT operator to select rows that are not within a given range. In this case, vendors with zip codes that aren't between 93600 and 93799 are included in the results.

The third example shows how you can use a calculated value in the test expression. Here, the PaymentTotal and CreditTotal columns are subtracted from the InvoiceTotal column to give the balance due. Then, this value is compared to the range specified in the BETWEEN phrase.

The last example shows how you can use calculated values in the BETWEEN phrase. Here, the first value is the result of the GETDATE function, and the second value is the result of the GETDATE function plus 30 days. So the query results will include all those invoices that are due between the current date and 30 days from the current date.

The syntax of the WHERE clause with a BETWEEN phrase

```
WHERE test_expression [NOT] BETWEEN begin_expression AND end_expression
```

Examples of the BETWEEN phrase

A BETWEEN phrase with literal values

```
WHERE InvoiceDate BETWEEN '2008-05-01' AND '2008-05-31'
```

A BETWEEN phrase preceded by NOT

```
WHERE VendorZipCode NOT BETWEEN 93600 AND 93799
```

A BETWEEN phrase with a test expression coded as a calculated value

```
WHERE InvoiceTotal - PaymentTotal - CreditTotal BETWEEN 200 AND 500
```

A BETWEEN phrase with the upper and lower limits coded as calculated values

```
WHERE InvoiceDueDate BETWEEN GetDate() AND GetDate() + 30
```

Description

- You can use the BETWEEN phrase to test whether an expression falls within a range of values. The lower limit must be coded as the first expression, and the upper limit must be coded as the second expression. Otherwise, the result set will be empty.
- The two expressions used in the BETWEEN phrase for the range of values are inclusive. That is, the result set will include values that are equal to the upper or lower limit.
- You can use the NOT operator to test for an expression that's not within the given range.

Warning about date comparisons

- All columns that have the datetime data type include both a date and time, and so does the value returned by the GetDate function. But when you code a date literal like '2008-05-01', the time defaults to 00:00:00 on a 24-hour clock, or 12:00 AM (midnight). As a result, a date comparison may not yield the results you expect. For instance, May 31, 2008 at 2:00 PM isn't between '2008-05-01' and '2008-31-01'.
- To learn more about date comparisons, please see chapter 8.

Figure 3-13 How to use the BETWEEN operator

How to use the LIKE operator

One final operator you can use in a search condition is the LIKE operator shown in figure 3-14. You use this operator along with the *wildcards* shown at the top of this figure to specify a *string pattern*, or *mask*, you want to match. The examples shown in this figure illustrate how this works.

In the first example, the LIKE phrase specifies that all vendors in cities that start with the letters SAN should be included in the query results. Here, the percent sign (%) indicates that any characters can follow these three letters. So San Diego and Santa Ana are both included in the results.

The second example selects all vendors whose vendor name starts with the letters COMPU, followed by any one character, the letters ER, and any characters after that. Two vendor names that match that pattern are Compuserve and Computerworld.

The third example searches the values in the VendorContactLName column for a name that can be spelled two different ways: Damien or Damion. To do that, the mask specifies the two possible characters in the fifth position, E and O, within brackets.

The fourth example uses brackets to specify a range of values. In this case, the VendorState column is searched for values that start with the letter N and end with any letter from A to J. That excludes states like Nevada (NV) and New York (NY).

The fifth example shows how to use the caret (^) to exclude one or more characters from the pattern. Here, the pattern says that the value in the VendorState column must start with the letter N, but must not end with the letters K through Y. This produces the same result as the previous statement.

The last example in this figure shows how to use the NOT operator with a LIKE phrase. The condition in this example tests the VendorZipCode column for values that don't start with the numbers 1 through 9. The result is all zip codes that start with the number 0.

The LIKE operator provides a powerful technique for finding information in a database that can't be found using any other technique. Keep in mind, however, that this technique requires a lot of overhead, so it can reduce system performance. For this reason, you should avoid using the LIKE operator in production SQL code whenever possible.

If you need to search the text that's stored in your database, a better option is to use the *Integrated Full-Text Search* (*iFTS*) feature that's provided by SQL Server 2008. This feature provides more powerful and flexible ways to search for text, and it performs more efficiently than the LIKE operator. However, iFTS is an advanced feature that requires some setup and administration and is too complex to explain here. For more information, you can look up "full-text search" in Books Online.

The syntax of the WHERE clause with a LIKE phrase

```
WHERE match_expression [NOT] LIKE pattern
```

Wildcard symbols

Symbol	Description
%	Matches any string of zero or more characters.
_	Matches any single character.
[ ]	Matches a single character listed within the brackets.
[-]	Matches a single character within the given range.
[^]	Matches a single character not listed after the caret.

WHERE clauses that use the LIKE operator

Example	Results that match the mask
`WHERE VendorCity LIKE 'SAN%'`	"San Diego" and "Santa Ana"
`WHERE VendorName LIKE 'COMPU_ER%'`	"Compuserve" and "Computerworld"
`WHERE VendorContactLName LIKE 'DAMI[EO]N'`	"Damien" and "Damion"
`WHERE VendorState LIKE 'N[A-J]'`	"NC" and "NJ" but not "NV" or "NY"
`WHERE VendorState LIKE 'N[^K-Y]'`	"NC" and "NJ" but not "NV" or "NY"
`WHERE VendorZipCode NOT LIKE '[1-9]%'`	"02107" and "08816"

Description

- You use the LIKE operator to retrieve rows that match a *string pattern*, called a *mask*. Within the mask, you can use special characters, called *wildcards*, that determine which values in the column satisfy the condition.
- You can use the NOT keyword before the LIKE keyword. Then, only those rows with values that don't match the string pattern will be included in the result set.
- Most LIKE phrases will significantly degrade performance compared to other types of searches, so use them only when necessary.

Figure 3-14 How to use the LIKE operator

How to use the IS NULL clause

In chapter 1, you learned that a column can contain a *null value*. A null isn't the same as zero, a blank string that contains one or more spaces (' '), or an empty string (''). Instead, a null value indicates that the data is not applicable, not available, or unknown. When you allow null values in one or more columns, you need to know how to test for them in search conditions. To do that, you can use the IS NULL clause as shown in figure 3-15.

This figure uses a table named NullSample to illustrate how to search for null values. This table contains two columns. The first column, InvoiceID, is an identity column. The second column, InvoiceTotal, contains the total for the invoice, which can be a null value. As you can see in the first example, the invoice with InvoiceID 3 contains a null value.

The second example in this figure shows what happens when you retrieve all the invoices with invoice totals equal to zero. Notice that the row that has a null invoice total isn't included in the result set. Likewise, it isn't included in the result set that contains all the invoices with invoices totals that aren't equal to zero, as illustrated by the third example. Instead, you have to use the IS NULL clause to retrieve rows with null values, as shown in the fourth example.

You can also use the NOT operator with the IS NULL clause as illustrated in the last example in this figure. When you use this operator, all of the rows that don't contain null values are included in the query results.

The syntax of the WHERE clause with the IS NULL clause

```
WHERE expression IS [NOT] NULL
```

The contents of the NullSample table

```
SELECT *
FROM NullSample
```

	InvoiceID	InvoiceTotal
1	1	125.00
2	2	0.00
3	3	NULL
4	4	2199.99
5	5	0.00

A SELECT statement that retrieves rows with zero values

```
SELECT *
FROM NullSample
WHERE InvoiceTotal = 0
```

	InvoiceID	InvoiceTotal
1	2	0.00
2	5	0.00

A SELECT statement that retrieves rows with non-zero values

```
SELECT *
FROM NullSample
WHERE InvoiceTotal <> 0
```

	InvoiceID	InvoiceTotal
1	1	125.00
2	4	2199.99

A SELECT statement that retrieves rows with null values

```
SELECT *
FROM NullSample
WHERE InvoiceTotal IS NULL
```

	InvoiceID	InvoiceTotal
1	3	NULL

A SELECT statement that retrieves rows without null values

```
SELECT *
FROM NullSample
WHERE InvoiceTotal IS NOT NULL
```

	InvoiceID	InvoiceTotal
1	1	125.00
2	2	0.00
3	4	2199.99
4	5	0.00

Description

- A *null value* represents a value that's unknown, unavailable, or not applicable. It isn't the same as a zero, a blank space (' '), or an empty string ('').
- To test for a null value, you can use the IS NULL clause. You can also use the NOT keyword with this clause to test for values that aren't null.
- The definition of each column in a table indicates whether or not it can store null values. Before you work with a table, you should identify those columns that allow null values so you can accommodate them in your queries.

Note

- SQL Server provides an extension that lets you use = NULL to test for null values. For this to work, however, the ANSI_NULLS system option must be set to OFF. For more information on this option, see Books Online.

Figure 3-15 How to use the IS NULL clause

How to code the ORDER BY clause

The ORDER BY clause specifies the sort order for the rows in a result set. In most cases, you can use column names from the base table to specify the sort order as you saw in some of the examples earlier in this chapter. However, you can also use other techniques to sort the rows in a result set, as described in the topics that follow.

How to sort a result set by a column name

Figure 3-16 presents the expanded syntax of the ORDER BY clause. As you can see, you can sort by one or more expressions in either ascending or descending sequence. This is illustrated by the three examples in this figure.

The first two examples show how to sort the rows in a result set by a single column. In the first example, the rows in the Vendors table are sorted in ascending sequence by the VendorName column. Since ascending is the default sequence, the ASC keyword is omitted. In the second example, the rows are sorted by the VendorName column in descending sequence.

To sort by more then one column, you simply list the names in the ORDER BY clause separated by commas as shown in the third example. Here, the rows in the Vendors table are first sorted by the VendorState column in ascending sequence. Then, within each state, the rows are sorted by the VendorCity column in ascending sequence. Finally, within each city, the rows are sorted by the VendorName column in ascending sequence. This can be referred to as a *nested sort* because one sort is nested within another.

Although all of the columns in this example are sorted in ascending sequence, you should know that doesn't have to be the case. For example, I could have sorted by the VendorName column in descending sequence like this:

```
ORDER BY VendorState, VendorCity, VendorName DESC
```

Note that the DESC keyword in this example applies only to the VendorName column. The VendorState and VendorCity columns are still sorted in ascending sequence.

The expanded syntax of the ORDER BY clause

```
ORDER BY expression [ASC|DESC] [, expression [ASC|DESC]] ...
```

An ORDER BY clause that sorts by one column in ascending sequence

```
SELECT VendorName,
    VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY VendorName
```

	VendorName	Address
1	Abbey Office Furnishings	Fresno, CA 93722
2	American Booksellers Assoc	Tarrytown, NY 10591
3	American Express	Los Angeles, CA 90096

An ORDER BY clause that sorts by one column in descending sequence

```
SELECT VendorName,
    VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY VendorName DESC
```

	VendorName	Address
1	Zylka Design	Fresno, CA 93711
2	Zip Print & Copy Center	Fresno, CA 93777
3	Zee Medical Service Co	Washington, IA 52353

An ORDER BY clause that sorts by three columns

```
SELECT VendorName,
    VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY VendorState, VendorCity, VendorName
```

	VendorName	Address
1	AT&T	Phoenix, AZ 85062
2	Computer Library	Phoenix, AZ 85023
3	Wells Fargo Bank	Phoenix, AZ 85038
4	Aztek Label	Anaheim, CA 92807
5	Blue Shield of California	Anaheim, CA 92850
6	Diversified Printing & Pub	Brea, CA 92621
7	Abbey Office Furnishings	Fresno, CA 93722
8	ASC Signs	Fresno, CA 93703
9	BFI Industries	Fresno, CA 93792

Description

- The ORDER BY clause specifies how you want the rows in the result set sorted. You can sort by one or more columns, and you can sort each column in either ascending (ASC) or descending (DESC) sequence. ASC is the default.
- By default, in an ascending sort, nulls appear first in the sort sequence, followed by special characters, then numbers, then letters. Although you can change this sequence, that's beyond the scope of this book.
- You can sort by any column in the base table regardless of whether it's included in the SELECT clause. The exception is if the query includes the DISTINCT keyword. Then, you can only sort by columns included in the SELECT clause.

Figure 3-16 How to sort a result set by a column name

How to sort a result set by an alias, an expression, or a column number

Figure 3-17 presents three more techniques you can use to specify sort columns. First, you can use a column alias that's defined in the SELECT clause. The first SELECT statement in this figure, for example, sorts by a column named Address, which is an alias for the concatenation of the VendorCity, VendorState, and VendorZipCode columns. Within the Address column, the result set is also sorted by the VendorName column.

You can also use an arithmetic or string expression in the ORDER BY clause, as illustrated by the second example in this figure. Here, the expression consists of the VendorContactLName column concatenated with the VendorContactFName column. Here, neither of these columns is included in the SELECT clause. Although SQL Server allows this seldom-used coding technique, many other database systems do not.

The last example in this figure shows how you can use column numbers to specify a sort order. To use this technique, you code the number that corresponds to the column of the result set, where 1 is the first column, 2 is the second column, and so on. In this example, the ORDER BY clause sorts the result set by the second column, which contains the concatenated address, then by the first column, which contains the vendor name. The result set returned by this statement is the same as the result set returned by the first statement. Notice, however, that the statement that uses column numbers is more difficult to read because you have to look at the SELECT clause to see what columns the numbers refer to. In addition, if you add or remove columns from the SELECT clause, you may also have to change the ORDER BY clause to reflect the new column positions. As a result, you should avoid using this technique.

An ORDER BY clause that uses an alias

```
SELECT VendorName,
    VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY Address, VendorName
```

	VendorName	Address
1	Aztek Label	Anaheim, CA 92807
2	Blue Shield of California	Anaheim, CA 92850
3	Malloy Lithographing Inc	Ann Arbor, MI 48106

An ORDER BY clause that uses an expression

```
SELECT VendorName,
    VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY VendorContactLName + VendorContactFName
```

	VendorName	Address
1	Dristas Groom & McCormick	Fresno, CA 93720
2	Internal Revenue Service	Fresno, CA 93888
3	US Postal Service	Madison, WI 53707

An ORDER BY clause that uses column positions

```
SELECT VendorName,
    VendorCity + ', ' + VendorState + ' ' + VendorZipCode AS Address
FROM Vendors
ORDER BY 2, 1
```

	VendorName	Address
1	Aztek Label	Anaheim, CA 92807
2	Blue Shield of California	Anaheim, CA 92850
3	Malloy Lithographing Inc	Ann Arbor, MI 48106

Description

- The ORDER BY clause can include a column alias that's specified in the SELECT clause.
- The ORDER BY clause can include any valid expression. The expression can refer to any column in the base table, even if it isn't included in the result set.
- The ORDER BY clause can use numbers to specify the columns to use for sorting. In that case, 1 represents the first column in the result set, 2 represents the second column, and so on.

Figure 3-17 How to sort a result set by an alias, an expression, or a column number

Perspective

The goal of this chapter has been to teach you the basic skills for coding SELECT statements. You'll use these skills in almost every SELECT statement you code. As you'll see in the chapters that follow, however, there's a lot more to coding SELECT statements than what's presented here. In the next three chapters, then, you'll learn additional skills for coding SELECT statements. When you complete those chapters, you'll know everything you need to know about retrieving data from a SQL Server database.

Terms

keyword
filter
Boolean expression
predicate
expression
column alias
substitute name
string expression
concatenate
concatenation operator
literal value
string literal
string constant
arithmetic expression
arithmetic operator
order of precedence
function
parameter
argument
date literal
comparison operator
logical operator
compound condition
subquery
string pattern
mask
wildcard
Integrated Full-Text Search (iFTS)
null value
nested sort

Exercises

1. Write a SELECT statement that returns three columns from the Vendors table: VendorContactFName, VendorContactLName, and VendorName. Sort the result set by last name, then by first name.
2. Write a SELECT statement that returns four columns from the Invoices table, named Number, Total, Credits, and Balance:

Number	Column alias for the InvoiceNumber column
Total	Column alias for the InvoiceTotal column
Credits	Sum of the PaymentTotal and CreditTotal columns
Balance	InvoiceTotal minus the sum of PaymentTotal and CreditTotal

   a. Use the AS keyword to assign column aliases.
   b. Use the = assignment operator to assign column aliases.
3. Write a SELECT statement that returns one column from the Vendors table named Full Name. Create this column from the VendorContactFName and VendorContactLName columns. Format it as follows: last name, comma, first name (for example, "Doe, John"). Sort the result set by last name, then by first name.
4. Write a SELECT statement that returns three columns:

InvoiceTotal	From the Invoices table
10%	10% of the value of InvoiceTotal
Plus 10%	The value of InvoiceTotal plus 10%

   (For example, if InvoiceTotal is 100.0000, 10% is 10.0000, and Plus 10% is 110.0000.) Only return those rows with a balance due greater than 1000. Sort the result set by InvoiceTotal, with the largest invoice first.
5. Modify the solution to exercise 2a to filter for invoices with an InvoiceTotal that's greater than or equal to $500 but less than or equal to $10,000.
6. Modify the solution to exercise 3 to filter for contacts whose last name begins with the letter A, B, C, or E.
7. Write a SELECT statement that determines whether the PaymentDate column of the Invoices table has any invalid values. To be valid, PaymentDate must be a null value if there's a balance due and a non-null value if there's no balance due. Code a compound condition in the WHERE clause that tests for these conditions.