The SELECT Statement

The SELECT statement is used primarily to retrieve specific data. A SELECT statement can be simple or complex—complex is not necessarily better. Try to make your SELECT statements as simple as possible while still retrieving the results you need. For example, if you need data from only two columns of a table, include only those two columns in the SELECT statement to minimize the amount of data that must be returned.

After you have decided what data you need from which tables, you can determine which other options, if any, you should use. These options can include specifying which columns should be in the WHERE clause to make use of your indexes, specifying whether the returned data should be sorted, and specifying whether you want only distinct values returned.

Let's begin by examining the various options for the SELECT statement and reviewing examples of each. The sample databases used in these examples, pubs and Northwind, will be needed to run these examples. To familiarize yourself with the pubs and Northwind databases, use SQL Server Management Studio to examine these database tables.

The syntax for the SELECT statement consists of several clauses, most of which are optional. A SELECT statement must include at least a SELECT clause and a FROM clause. These two clauses identify which column or columns of data to retrieve and from which table or tables to retrieve the data, respectively. For example, a simple SELECT statement to retrieve the authors' first and last names from the authors table in the pubs database might look like this:

```
SELECT   au_fname, au_lname
FROM     authors

SELECT   au_fname, au_lname
FROM     authors
```

> **NOTE**
>
> Because keywords are not case sensitive, you can use any capitalization system you want. It's a good idea to be consistent just to make your code easier to read. For this reason, the examples in these notes use uppercase letters for keywords.

When you run the SELECT statement interactively—for example, using SQL Query Analyzer - the results are displayed in columns, with column headings for clarity.

The SELECT Clause

The SELECT clause consists of a required select list and possibly some optional arguments. The *select list* is the list of expressions or columns that you specify in the SELECT clause to indicate which data should be returned. The optional arguments and the select list are described in this section.

Arguments

The following two arguments can be used in the SELECT clause to control which rows are returned:

- **DISTINCT** Returns only unique rows. If the select list contains several columns, the rows will be considered unique if the corresponding values in at least one of the columns differ. For two rows to be duplicates, they must contain identical values in every column.
- **TOP *n* [PERCENT]** Returns only the first *n* rows from the result set. If PERCENT is specified, only the first *n* percent of the rows are returned. When PERCENT is used, *n* must be between *0* and *100*. If the query includes an ORDER BY clause, the rows are ordered first and then the first *n* or first n percent are returned from the ordered result set. (ORDER BY clauses are described in the section **The ORDER BY Clause** a bit later)

The following T-SQL code shows our sample SELECT statement run three times, each time with a different argument. The first query uses the DISTINCT argument, the second query uses the TOP 50 PERCENT argument, and the third query uses the TOP 5 argument.

```
SELECT   DISTINCT au_fname, au_lname
FROM     authors

SELECT   TOP 50 PERCENT au_fname, au_lname
FROM     authors

SELECT   TOP 5 au_fname, au_lname
FROM     authors
```

The first query returns 23 rows, each of which is unique. The second query returns 12 rows (approximately 50 percent, rounded up), and the third query returns 5 rows.

The Select List

As mentioned, the select list is the list of expressions or columns that you specify in the SELECT clause to indicate which data should be returned. An expression can be a list of column names, functions, or constants. The select list can include several expressions or column names, separated by commas. The preceding examples use the following select list:

```
au_fname, au_lname
```

The *, or Wildcard Character You can use the asterisk (*), or wildcard character, in the select list to return all columns from all tables and views named in the FROM clause of the query. For example, to return all columns of all rows from the *sales* table in the pubs database, use the following query:

```
SELECT   *
FROM     sales
GO
```

The section **Cross Joins** later describes what happens when more than one table is listed in the FROM clause of a SELECT statement that contains the wildcard character.

Column Aliases Using a column alias in the select list allows you to specify the column heading that you want to appear in the result set. You can use an alias to clarify the meaning of the data in an output column, to assign a heading to a column that is used in a function, and to refer to an ORDER BY clause.

When two or more columns with the same name exist in different tables, you might want to include the table name in the column heading of the output for clarity. For an example using a column alias, let's look at the *lname* column in the *employee* table of the pubs database. You could issue the following query:

```
SELECT   lname
FROM     employee
```

If you made such a query, you would get the following results:

```
lname
--------
Cruz
Roulet

Devon
O'Rourke
Ashworth
Latimer

(43 rows affected)
```

To display the heading "Employee Last Name" instead of the original heading, *lname*, in the result set (to emphasize the fact that the last name is from the *employee* table), use the AS keyword, as shown here:

```
SELECT   lname AS "Employee Last Name"
FROM     employee
```

The output from this command is shown here:

```
Employee Last Name
------------------
Cruz
Roulet

Devon
O'Rourke
Ashworth
Latimer

(43 rows affected)
```

You can also use a column alias with other types of expressions in the select list and as a reference column in an ORDER BY clause. Suppose you have a function call in the select list. To assign a column alias that describes the output from the function, use the AS keyword after the function call. If you do not use an alias with a function, there will be no column heading at all. For example, the following statement assigns the column heading "Maximum Job ID" for the output of the MAX function:

```
SELECT   MAX(job_id) AS "Maximum Job ID"
FROM     employee
```

The column alias is enclosed in quotation marks because it contains multiple words with spaces between them. If the alias does not include spaces, you do not have to enclose it in quotation marks, as you'll see in the next example.

You can reference a column alias that was assigned in the SELECT clause as an argument in the ORDER BY clause. This technique is useful when the select list contains a function whose results need to be sorted. For example, the following command retrieves the quantity of books sold at each store and sorts the output by quantity. The alias assigned in the select list is used in the ORDER BY clause.

```
SELECT   SUM(qty) AS Quantity_of_Books, stor_id
FROM     sales
GROUP BY stor_id
ORDER BY Quantity_of_Books
```

In this case, we did not enclose the alias in quotation marks because it contains no spaces.

If we had not assigned a column alias for *SUM(qty)* in this query, we could have used *SUM(qty)* instead of the alias in the ORDER BY clause. This technique, shown in the next example, will provide the same output, but with no column heading for the *sum* column:

```
SELECT   SUM(qty), stor_id FROM sales
GROUP BY stor_id
ORDER BY SUM(qty)
```

Remember that a column alias is used to assign a heading to a column for output purposes; it does not affect the results of the query in any way.

The FROM Clause

The FROM clause contains the names of the tables and views from which the data is pulled. Every SELECT statement requires a FROM clause, except when the select list contains no column names—only constants, variables, and arithmetic expressions. You've already seen some simple examples of the FROM clause, but FROM clauses can also contain derived tables, joins, and aliases.

Derived Tables

A *derived table* is the result set from a SELECT statement nested in the FROM clause. The result set of the nested SELECT statement is used as a table from which the outer SELECT statement selects its data. The following query uses a derived table to find the names of any stores that honor at least one type of discount:

```
SELECT   s.stor_name
FROM     stores AS s, (SELECT stor_id, COUNT(DISTINCT discounttype)
                       AS d_count
                       FROM     discounts
                       GROUP BY stor_id) AS d
WHERE    s.stor_id = d.stor_id AND
         d.d_count >= 1
```

If you run this command, you should see one row selected, which means that only one store in the database, Bookbeat, offers any discount.

Notice that this query uses shorthand for the table names (*s* for the *stores* table and *d* for the *discounts* table). This shorthand, called a *table alias*, is described in the section **Table Aliases** later.

Joined Tables

A *joined table* is a result set from the join operation performed on two or more tables. Several types of joins can be performed on tables: inner joins, full outer joins, left outer joins, right outer joins, and cross joins. Let's look at each of these joins in detail.

Inner Joins An *inner join* is the default join type; it specifies that only table rows matching the ON condition should be included in the result set and that any unmatched rows should be discarded. To specify a join, use the JOIN keyword. Use the ON keyword to identify the search condition on which to base the join. The following query joins the *stores* and *discounts* tables to show which stores offer a discount and the type of discount. (By default, this is an inner join, which means that only rows matching the ON search condition are returned.)

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

The result set looks like this:

```
stor_id  discounttype
-------  --------------------
8042     Customer Discount
```

As you can see, only one store offers a discount, and it has only one type of discount. The only row returned is the one whose *stor_id* from the *stores* table has a matching *stor_id* from the *discounts* table. That particular *stor_id* and its associated discounttype are returned.

Full Outer Joins A *full outer join* specifies that the unmatched rows (rows that do not meet the ON condition) as well as the matched rows (rows that meet the ON condition) should be included in the result set. For unmatched rows, *NULL* will appear in the column that did not match. In this example, *NULL* means either that a store did not offer any discount, and thus it has a *stor_id* value in the *stores* table but not in the *discounts* table, or that a type of discount in the *discounts* table is not offered by any store. The following query uses the same query as the preceding inner join, but this time, we will specify FULL OUTER JOIN:

```
SELECT   s.stor_id, d.discounttype
FROM     stores s FULL OUTER JOIN discounts d
ON       s.stor_id = d.stor_id
```

The result set looks like this:

```
stor_id  discounttype
-------  ------------------
NULL     Initial Customer
NULL     Volume Discount
6380     NULL
7066     NULL
7067     NULL
7131     NULL
7896     NULL
8042     Customer Discount
```

Only one of the results rows shows a match—the last row. The other rows have *NULL* in one column.

Left Outer Joins A *left outer join* returns the matching rows plus all the rows from the table that is specified to the left of the JOIN keyword. Using the same query, we specify LEFT OUTER JOIN this time, as shown here:

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

The result set looks like this:

```
stor_id discounttype
------- ----------------------------------------
6380    NULL
7066    NULL
7067    NULL
7131    NULL
7896    NULL
8042    Customer Discount
```

This result set includes the rows from the *stores* table that had no matching *stor_id* value in the *discounts* table. (The *discounttype* column for those rows is *NULL*.) The result set also includes the one row that matched the ON condition.

Right Outer Joins A *right outer join* is the opposite of a left outer join: it returns the matching rows plus all the rows from the table specified to the right of the JOIN keyword. Here is the same query with RIGHT OUTER JOIN specified:

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

The result set looks like this:

```
stor_id  discounttype
-------  --------------------
NULL     Initial Customer
NULL     Volume Discount
8042     Customer Discount
```

This result set shows the rows from the *discounts* table that do not have a matching *stor_id* value in the *stores* table. (The *stor_id* column for those rows is *NULL*.) The result set also shows the one row that matched the ON condition.

Cross Joins A *cross join* is the product of two tables when no WHERE clause is specified. When a WHERE clause is specified, the cross join acts like an inner join. Without a WHERE clause, all rows and columns will be returned from both tables in the following manner: each row from the first table will be matched with each row from the second table, so the size of the result set will be the number of rows in the first table multiplied by the number of rows in the second table.

To understand a cross join, let's start with some new examples. First we'll look at a cross join without a WHERE clause, and then we'll look at three examples of cross joins that include WHERE clauses. The following queries show a simple example. Run the three queries and note the number of rows that result from each.

```
SELECT   *
FROM     stores
SELECT   *
FROM     sales
SELECT   *
FROM     stores CROSS JOIN sales
```

NOTE

If you include two tables in the FROM clause, the effect is the same as specifying CROSS JOIN, as in the following example:

```
SELECT   *
FROM     stores, sales
```

To avoid this jumble of information (if it is more than we need), we can add a WHERE clause to narrow the query, as in the following statement:

```
SELECT   *
FROM     sales CROSS JOIN stores
WHERE    sales.stor_id = stores.stor_id
```

This statement returns only the rows that match the search condition in the WHERE clause, which narrows the result set to 21 rows. The WHERE clause forces a cross join to act the same as an inner join. (That is, only rows matching the search condition are returned.) The preceding query returns the rows in the *sales* table, concatenated with the rows from the *stores* table that have the same *stor_id* value. Rows that do not contain a match are not returned.

To further narrow the result set, you can specify from which table to select all rows and columns by adding the table name before the asterisk (*), as in the following query. You can also specify to which table a column belongs by inserting the table name and a dot (.) before any column name.

```
SELECT   sales.*, stores.city
FROM     sales CROSS JOIN stores
WHERE    sales.stor_id = stores.stor_id
```

This query returns all the columns from the *sales* table, with the city column from the *stores* table row that has the same *stor_id* value appended. In effect, the result set includes the city of the store where the sale was made appended to the rows in the *sales* table that have a matching *stor_id* value in the stores table.

Here is the same query without the * symbol; only the *stor_id* column will be selected from the *sales* table:

```
SELECT   sales.stor_id, stores.city
FROM     sales CROSS JOIN stores
WHERE    sales.stor_id = stores.stor_id
```

Table Aliases

We've already looked at several examples in which a table name alias was used. Specifying the AS keyword is optional. (FROM tablename AS alias gives the same result as FROM tablename alias.) Let's look again at the query from the "Right Outer Joins" section, which used aliases:

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

Each of the two tables has a stor_id column. To distinguish which table's stor_id column you are referring to in the query, you must supply the table name or an alias followed by a dot (.) and then the column name. In this example, the alias s is used for the stores table, and d is used for the discounts table. When specifying a column, we must add s. or d. before the column name to indicate which table contains it. The same query with the AS keyword included looks like this:

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

The WHERE Clause and Search Conditions

You can use the WHERE clause to restrict the rows that are returned from a query, according to the search conditions specified. In this section, we'll examine many of the operations that can be used in the search condition.

> NOTE
>
> Search conditions are used not only in WHERE clauses for the SELECT statement, but they are also used in UPDATE and DELETE statements. (The UPDATE and DELETE statements will be covered later.)

First let's review some terminology. The search condition can contain an unlimited number of predicates joined by the logical operators AND, OR, and NOT. A *predicate* is an expression that returns a value of TRUE, FALSE, or UNKNOWN. An *expression* can be a column name, a constant, a scalar function (a function that returns one value), a variable, a scalar subquery (a subquery that returns one column), or a combination of these elements joined by operators. In this section, the term "expression" refers to predicates and expressions.

Comparison Operators

The equality and nonequality operators that can be used with expressions are listed in the table below.

Table: *Comparison operators*

Operator	Condition Tested
=	Tests for equality between two expressions
<>	Tests whether two expressions are not equal to each other
!=	Tests whether two expressions are not equal to each other (same as <>)
>	Tests whether one expression is greater than the other

>=	Tests whether one expression is greater than or equal to the other
!>	Tests whether one expression is not greater than the other
<	Tests whether one expression is less than the other
<=	Tests whether one expression is less than or equal to the other
!<	Tests whether one expression is not less than the other

A simple WHERE clause might compare two expressions by using the equality operator (=). For example, the following SELECT statement tests the value in the *lname* column for each row, which is of the char data type, and returns TRUE if the value is equal to "*Latimer*." (The rows that return TRUE will be included in the result set.)

```
SELECT    *
FROM      employee
WHERE     lname = "Latimer"
```

In this case, the query returns one row. The name *Latimer* must be enclosed in quotation marks because it is a character string.

The following query uses the not equal operator (<>), this time with an *integer* data type column, *job_id*:

```
SELECT    job_desc
FROM      jobs
WHERE     job_id <> 1
GO
```

This query will return the job description text from the row or rows in the *jobs* table that have a *job_id value* not equal to 1. In this case, 13 rows are returned. If a row has a value of *NULL*, it does not equal 1 or any other value, so rows with null values will be returned as well.

Logical Operators

The logical operators AND and OR test two expressions and return a Boolean value of *TRUE*, *FALSE*, or *UNKNOWN*, depending on the results from the two expressions. The NOT operator negates the Boolean value returned by an expression that follows it.

The following query uses two expressions in the WHERE clause with the AND logical operator:

```
SELECT   job_desc, min_lvl, max_lvl
FROM     jobs
WHERE    min_lvl >= 100 AND
         max_lvl <= 225
```

In the next query, an OR operation tests for publishers in either Washington, D.C. or Massachusetts. A row will be returned if either of the tests returns *TRUE* for that row.

```
SELECT   p.pub_name, p.state, t.title
FROM     publishers p, titles t
WHERE    p.state = "DC" OR
         p.state = "MA" AND
         t.pub_id = p.pub_id
```

This query returns 23 rows.

The NOT operation simply returns the negation of the value of the Boolean expression that follows it. For example, to return all book titles for which an author's royalties were not less than 20 percent, you could use the NOT operator in the following manner:

```
SELECT   t.title, r.royalty
FROM     titles t, roysched r
WHERE    t.title_id = r.title_id AND NOT
         r.royalty  < 20
```

This query returns the 18 titles for which royalties were equal to or greater than 20 percent.

Other Keywords

In addition to the operators described in the preceding sections, a variety of T-SQL keywords can be used in a search condition. The most commonly used keywords are explained in this section, and examples of their use are given.

LIKE The LIKE keyword indicates pattern matching in a search condition. *Pattern matching* is testing for a match between a match expression and the pattern specified in the search condition, using the following syntax:

```
<match_expression> LIKE <pattern>
```

If the match expression matches the pattern, a Boolean value of TRUE is returned. Otherwise, *FALSE* is returned. The match expression must be of the *character string*

data type. If it is not, SQL Server will convert it to the *character string* data type, if possible.

Patterns are really string expressions. A *string expression* is defined as a string of characters and wildcard characters. *Wildcard characters* are characters that take on special meanings when used in a string expression. Table below lists the wildcard characters that can be used in patterns.

Table : *T-SQL Wildcard characters*

Wildcard Character	Description
%	Percent symbol; matches a string of zero or more characters
_	Underscore; matches any single character
[]	Range wildcard character; matches any single character within the range or set, such as [m-p] or [mnop], meaning any of the characters m, n, o, or p
[^]	Not-in-range wildcard character; matches any single character not within the range or set, such as [^m-p] or [^mnop], meaning any character other than m, n, o, or p

To get a better understanding of using the LIKE keyword and wildcard characters, let's look at some examples. To find all last names in the *authors* table that begin with the letter "S," you could use the following query with the % wildcard character:

```
SELECT   au_lname
FROM     authors
WHERE    au_lname LIKE "S%"
```

The result set will look like this:

```
au_lname
————————
Smith
Straight
Stringer
```

In this query, "*S%*" means return all rows that contain a last name beginning with "S," followed by any number of characters.

To retrieve the information for an author whose ID starts with the number 724, knowing that each ID is formatted like a social security number (three digits, followed by a dash, followed by two digits, then another dash, and finally four digits), you could use the _ wildcard character, as follows:

```
SELECT   *
FROM     authors
WHERE    au_id LIKE "724-_ _-_ _ _ _"
```

The result set will contain two rows, with *au_id* values of *724-08-9931* and *724-80-9391*.

Now let's look at an example that uses the [] wildcard. To retrieve the last names of authors starting with "A" through "M," you could use the [] wildcard along with the % wildcard character, as shown here:

```
SELECT   au_lname
FROM     authors
WHERE    au_lname LIKE "[A-M]%"
```

The result set will contain 14 rows of names beginning with "A" through "M" (13, if you are using a case-sensitive sort order).

If we perform a similar query but use the [^] wildcard in place of the [] wildcard character, we will get rows that contain last names that start with letters other than "A" through "M," as shown here:

```
SELECT   au_lname
FROM     authors
WHERE    au_lname LIKE "[^A-M]%"
```

This query returns nine rows.

If you are using a case-sensitive sort order and you want to find all names that fall into a range without regard to case, you could use a query that checks for a lowercase or an uppercase first letter, as shown here:

```
SELECT   au_lname
FROM     authors
WHERE    au_lname LIKE "[A-M]%" OR
         au_lname LIKE "[a-m]%"
```

This result set will include the name "del Castillo," whereas a case-sensitive query that checked for only uppercase "A" through "M" would not.

The LIKE keyword can also be preceded by the NOT operator. NOT LIKE returns rows that do not match the condition specified. For example, to select titles that do not start with the word "The," you could use NOT LIKE in the following query:

```
SELECT   title
FROM     titles
WHERE    title NOT LIKE "The %"
```

This query returns 15 rows.

You can be creative when using the LIKE keyword. But be careful to test your queries to be sure they are returning the data you expect. If you leave out a NOT or a ^ character when you meant to include one, your result set will be the opposite of what you desired. Failing to include the % wildcard character when it is needed will cause incorrect results also. And remember that leading and trailing spaces are also matched exactly.

ESCAPE The ESCAPE keyword enables you to perform pattern matching for the wildcard characters themselves, such as ^, %, [, and _. Following the ESCAPE keyword, you specify the character you want to use as the escape character, which signals that the following character in the string expression should be matched literally. For example, to search for all rows in the *titles* table that have an underscore in the *title* column, you would use the following query:

```
SELECT   title
FROM     titles
WHERE    title LIKE "%e_%" ESCAPE "e"
```

This query returns no rows because no titles in the database include an underscore.

BETWEEN The BETWEEN keyword is always used with AND and specifies an inclusive range to test for in a search condition. The syntax is shown here:

```
<test_expression> BETWEEN <begin_expression> AND <end_expression>
```

The result of the search condition will be the Boolean value *TRUE* if *test_expression* is greater than or equal to *begin_expression* and is also less than or equal to *end_expression*. Otherwise, the result will be *FALSE*.

The following query uses BETWEEN to find all the book titles that have a price between $5 and $25:

```
SELECT   price, title
FROM     titles
WHERE    price BETWEEN 5.00 AND 25.00
```

This query returns 14 rows.

You can also use NOT with BETWEEN to find rows that are not in the specified range. For example, to find the book titles whose prices are not between $20 and $30 (meaning that their prices are less than $20 or greater than $30), you would use the following query:

```
SELECT   price, title
FROM     titles
WHERE    price NOT BETWEEN 20.00 AND 30.00
```

When you use the BETWEEN keyword, *test_expression* must have the same data type as *begin_expression* and *end_expression*.

In the preceding example, the *price* column has the data type *money*, so *begin_expression* and *end_expression* must each be a number that can be compared with or implicitly converted to the *money* data type. You could not use *price* as *test_expression* and then use a character string (of the *char* data type) for *begin_expression* and *end_expression*. If you did, SQL Server would return an error message.

Our last example involving the BETWEEN keyword uses strings in a search condition. To find authors' last names that fall alphabetically between the names "Bennet" and "McBadden," you would use the following query:

```
SELECT   au_lname
FROM     authors
WHERE    au_lname BETWEEN "Bennet" AND "McBadden"
```

Because the BETWEEN range is inclusive, the results of this query will include the names "Bennet" and "McBadden," which do exist in the table.

IS NULL The IS NULL keyword is used in a search condition to select rows that have a null value in the specified column. For example, to find the book *titles* in the titles table that have no data in the *notes* column (that is, the value for *notes* is *NULL*), you would use the following query:

```
SELECT   title, notes
FROM     titles
WHERE    notes IS NULL
```

The result set looks like this:

```
title                                notes
———————————————————————————————   ————
The Psychology of Computer Cooking   NULL
```

As you can see, the null value in the *notes* column appears as *NULL* in the result set. *NULL* is not the actual value in the column—it simply indicates that a null value exists in that column. (Recall a null value is an unknown value.)

To find the titles that do have data in the *notes* column (titles for which the value of *notes* is not a null value), use IS NOT NULL, as follows:

```
SELECT   title, notes
FROM     titles
WHERE    notes IS NOT NULL
```

All of the 17 rows in the result set will have one or more characters in the *notes* column and therefore do not have null values in the *notes* column.

IN

The IN keyword is used in a search condition to determine whether the given test expression matches any value in a subquery or list of values. If a match is found, a value of *TRUE* is returned. NOT IN returns the negation of the result for IN, and therefore, if the test expression is not found in the subquery or the list of values, *TRUE* is returned. The syntax is as follows:

```
<test_expression> IN (<subquery>)
```

or

```
<test_expression> IN (<list of values>)
```

A *subquery* is a SELECT statement that returns only one column in the result set. The subquery must be enclosed in parentheses. A *list of values* is just that, with the values enclosed in parentheses and separated by commas. The column resulting from either the subquery or the list of values must have the same data type as *test_expression*. SQL Server will perform implicit conversion when necessary.

You could use IN with a list of values to find the job ID numbers of three specific job descriptions, as in the following query:

```
SELECT   job_id
FROM     jobs
WHERE    job_desc IN ("Operations Manager",
                      "Marketing Manager",
                      "Designer")
```

The list of values in this query is as follows: ("*;Operations Manager*", "*Marketing Manager*", "*Designer*"). The query returns the job IDs from rows that have one of these three values in the *job_desc* column. The IN keyword makes your query simpler and easier to read and understand than if you had used two OR operators, as shown here:

```
SELECT   job_id
FROM     jobs
WHERE    job_desc = "Operations Manager" OR
         job_desc = "Marketing Manager"  OR
         job_desc = "Designer"
```

The following query uses the IN keyword twice in one statement-once for a subquery and once for a list of values within the subquery:

```
SELECT   fname, lname    —Outer query
FROM     employee
WHERE    job_id IN ( SELECT job_id    —Inner query, or subquery
                     FROM   jobs
                     WHERE  job_desc IN ("Operations Manager",
                                         "Marketing Manager",
                                         "Designer"))
```

The subquery result set is found first-in this case, a set of *job_id* values. The *job_id* values resulting from the subquery are not returned to the screen; the outer query uses them as the expression for its own IN search condition. The final result set will contain the first and last names of all employees whose job titles are Operations Manager, Marketing Manager, or Designer. Here is the result set:

```
fname                lname
———————————————— ————————————————————————
Pedro                Afonso
Lesley               Brown
Palle                Ibsen
Karin                Josephs
Maria                Larsson
Elizabeth            Lincoln
Patricia             McKenna
Roland               Mendel
Helvetius            Nagy
Miguel               Paolino
Daniel               Tonini
(11 rows affected)
```

IN can also be used with the NOT operator. For example, to return the names of all publishers not located in California, Texas, or Illinois, run the following query:

```
SELECT   pub_name
FROM     publishers
WHERE    state NOT IN ("CA",
                       "TX",
                       "IL")
```

This query will return five rows whose *state* column value is not one of the three states in the list of values. If you have the database option *ANSI nulls* set to *ON*, the result set will contain only three rows. This reduction is because two of the five rows from the original result set will have *NULL* as the *state* value, and NULLs are not selected when *ANSI nulls* is set to ON.

To determine your *ANSI nulls* setting for the pubs database, run the following system stored procedure:

```
sp_dboption "pubs", "ANSI nulls"
```

If *ANSI nulls* is set to *OFF*, change the value to *ON* by using the following statement:

```
sp_dboption "pubs", "ANSI nulls", TRUE
```

To change the value from *ON* to *OFF*, use *FALSE* in place of *TRUE.*

EXISTS The EXISTS keyword is used to test for the existence of rows in the subquery that follows. The syntax is shown here:

```
EXISTS (<subquery>)
```

If any rows satisfy the subquery, TRUE is returned.

To select names of authors who have already published a book, you could use the following query:

```
SELECT   au_fname, au_lname
FROM     authors
WHERE    EXISTS (SELECT au_id
                 FROM   titleauthor
                 WHERE  titleauthor.au_id = authors.au_id)
```

Authors whose names are in the *authors* table but who have not published a book listed in the *titleauthor* table will not be selected. If no rows had been selected in the subquery, the result set for the outer query would be empty. (Zero rows would be selected.)

The GROUP BY Clause

GROUP BY is used after the WHERE clause to indicate that the result set rows should be grouped according to the grouping columns specified. If an aggregate function is used in the SELECT clause, an aggregate summary value is calculated for each group and shown in the output. (An *aggregate function* performs a calculation and returns a value; these functions are described in detail in the section **Aggregate Functions** later.)

> NOTE
>
> Every column in the select list-except for columns used in an aggregate function-must be specified in the GROUP BY clause as a grouping column; otherwise, SQL Server will return an error message.

GROUP BY is most useful when an aggregate function is included in the SELECT clause. Let's take a look at a SELECT statement that uses the GROUP BY clause to find the total number sold of each book title:

```
SELECT   title_id, SUM(qty)
FROM     sales
GROUP BY title_id
```

The result set looks like this:

```
title_id
------ --------
BU1032           15
BU1111           25
BU2075           35
BU7832           15
MC2222           10
MC3021           40
PC1035           30
PC8888           50
PS1372           20
PS2091          108
PS2106           25
```

```
PS3333            15
PS7777            25
TC3218            40
TC4203            20
TC7777            20
(16 rows affected)
```

This query does not contain a WHERE clause-you do not need one. The result set shows a *title_id* column and a summary column with no heading. For each distinct title ID, the total number sold of that title appears in the summary column. For example, the *title_id* value *BU1032* appears twice in the *sales* table-it appears once showing 5 sales in the *qty* column, and it appears again showing 10 sales for a different order. The SUM aggregate function adds these two sales to arrive at the total sales figure of 15, which appears in the summary column. To add a heading to your summary column, use the AS keyword, as shown here:

```
SELECT   title_id, SUM(qty) AS "Total Sales"
FROM     sales
GROUP BY title_id
```

Now the result set will show the heading "Total Sales" over the summary column:

```
title_id  Total Sales
--------  -----------
BU1032            15
BU1111            25
BU2075            35
BU7832            15
MC2222            10
MC3021            40
PC1035            30
PC8888            50
PS1372            20
PS2091           108
PS2106            25
PS3333            15
PS7777            25
TC3218            40
TC4203            20
TC7777            20
(16 rows affected)
```

You can nest groups by including more than one column in the GROUP BY clause. Nesting groups means that the result set will be grouped by each of the grouping

columns in the order in which the columns are specified. For example, to find the average price for book titles that are grouped by type and then by publisher, run the following query:

```
SELECT   type, pub_id, AVG(price) AS "Average Price"
FROM     titles
GROUP BY type, pub_id
```

The result set looks like this:

```
type         pub_id Average Price
———————————— ————— ————————————————————
business     0736                    2.99
psychology   0736                   11.48
UNDECIDED    0877                    NULL
mod_cook     0877                   11.49
psychology   0877                   21.59
trad_cook    0877                   15.96
business     1389                   17.31
popular_comp 1389                   21.48
(8 rows affected)
```

Notice that the psychology and business types occur more than once because they are grouped under different publisher IDs. The *NULL* average price for the UNDECIDED type reflects that no prices were inserted into the table for that type, and therefore, no average could be calculated.

GROUP BY provides an optional keyword, ALL, that specifies that all groups should be included in the result set, even if they do not meet the search condition. The groups that do not have rows that meet the search condition will contain a **NULL** in the summary column so that they can be easily identified. For example, to show the average price for books that have a royalty of 12 percent (and also show books that do not, which will have **NULL** in the summary column) and to group the books by type and then by publisher ID, run the following query:

```
SELECT   type, pub_id, AVG(price) AS "Average Price"
FROM     titles
WHERE    royalty = 12
GROUP BY ALL type, pub_id
```

The result set looks like this:

```
type         pub_id Average Price
———————————— ————— ————————————————
business     0736                NULL
psychology   0736               10.95
UNDECIDED    0877                NULL
mod_cook     0877               19.99
psychology   0877                NULL
trad_cook    0877                NULL
business     1389                NULL
popular_comp 1389                NULL
(8 rows affected)
```

All types are present in the output and *NULL* appears for the types that do not have a book with a commission of 12 percent.

If we now remove the keyword ALL, the result set will contain only types that have a book with a 12 percent commission, as shown here:

```
type         pub_id Average Price
———————————— ————— ————————————————
psychology   0736               10.95
mod_cook     0877               19.99
(2 rows affected)
```

The GROUP BY clause is often accompanied by the HAVING clause, which is covered next.

The HAVING Clause

The HAVING clause is used to specify a search condition for a group or an aggregate function. HAVING is most commonly used after a GROUP BY clause for cases in which a search condition must be tested after the results are grouped. If the search condition can be applied before the grouping occurs, it is more efficient to place the search condition in the WHERE clause than to add a HAVING clause. This technique reduces the number of rows that must be grouped. If there is no GROUP BY clause, HAVING can be used only with an aggregate function in the select list. In this case, the HAVING clause acts the same as a WHERE clause. If HAVING is not used in either of these ways, SQL Server will return an error message.

The syntax for the HAVING clause is as follows:

```
HAVING <search_condition>
```

Here, *search_condition* has the same meaning as the search conditions described in the section **The WHERE Clause and Search Conditions** earlier. One difference

between the HAVING clause and the WHERE clause is that the HAVING clause can include an aggregate function in the search condition, but the WHERE clause cannot.

> **NOTE**
>
> You can use aggregate functions in the SELECT clause and in the HAVING clause, but you can't use them in the WHERE clause.

The following query uses the HAVING clause to select the types of books per publisher that have an average price greater than $15:

```
SELECT   type, pub_id, AVG(price) AS "Average Price"
FROM     titles
GROUP BY type, pub_id
HAVING   AVG(price) > 15.00
```

The result set looks like this:

```
type         pub_id Average Price
———————————— ————— ————————————————————
psychology   0877                  21.59
trad_cook    0877                  15.96
business     1389                  17.31
popular_comp 1389                  21.48
(4 rows affected)
```

You can also use logical operators with the HAVING clause. Here, the AND operator has been added to our query:

```
SELECT   type, pub_id, AVG(price) AS "Average Price"
FROM     titles
GROUP BY type, pub_id
HAVING   AVG(price) >= 15.00 AND
         AVG(price) <= 20.00
```

The result set looks like this:

```
type         pub_id Average Price
———————————— ————— ————————————————————
trad_cook    0877                  15.96
business     1389                  17.31
(2 rows affected)
```

You could get the same results by using the BETWEEN clause instead of just AND, as shown here:

```
SELECT   type, pub_id, AVG(price) AS "Average Price"
FROM     titles
GROUP BY type, pub_id
HAVING   AVG(price) BETWEEN 15.00 AND 20.00
```

To use HAVING without a GROUP BY clause, you must have an aggregate function in the select list and in the HAVING clause. For example, to select the sum of the prices for books of type *mod_cook* only if the sum is greater than $20, run the following query:

```
SELECT   SUM(price)
FROM     titles
WHERE    type = "mod_cook"
HAVING   SUM(price) > 20
GO
```

If you try to put the expression *SUM(price) > 20* in the WHERE clause, SQL Server will return an error message. (Aggregate functions are not allowed in the WHERE clause.)

> **NOTE**
>
> Remember, the only time you can use the HAVING clause is when you add a search condition to test the resultant groups from a GROUP BY clause or to test an aggregate function. Otherwise, you should specify the search condition in the WHERE clause.

The ORDER BY Clause

The ORDER BY clause is used to specify the order in which the rows in a result set should be sorted. You can specify either ascending (from lowest to highest) or descending (from highest to lowest) order by using the keywords ASC or DESC. Ascending order is the default if no order is specified. You can specify more than one column in the ORDER BY clause. The results will be ordered by the first column listed. If the first column contains duplicate values, those rows will be ordered according to the second column listed, and so on. This ordering makes more sense when ORDER BY is used with GROUP BY, as you'll see later in this section. First let's look at an example that uses one column in the ORDER BY clause to list authors by last name, in ascending order:

```
SELECT   au_lname, au_fname
FROM     authors
ORDER BY au_lname ASC
```

The result set will be ordered alphabetically by last name. Remember that the case sensitivity of the sort order you set when installing SQL Server will affect how last names such as "del Castillo" will be ordered.

If you want to sort results on more than one column, simply add the column names, separated by commas, to the ORDER BY clause. The following query selects job IDs and employee first names and last names and then displays them ordered by job ID, then last name, and then first name:

```
SELECT   job_id, lname, fname
FROM     employee
ORDER BY job_ifd, lname, fname
```

The result set looks like this:

```
job_id lname                          fname
------ ------------------------------ --------------
     2 Cramer                         Philip
     3 Devon                          Ann
     4 Chang                          Francisco
     5 Henriot                        Paul
     5 Hernadez                       Carlos
     5 Labrune                        Janine
     5 Lebihan                        Laurence
     5 Muller                         Rita
     5 Ottlieb                        Sven
     5 Pontes                         Maria
     6 Ashworth                       Victoria
     6 Karttunen                      Matti
     6 Roel                           Diego
     6 Roulet                         Annette
     7 Brown                          Lesley
     7 Ibsen                          Palle
     7 Larsson                        Maria
     7 Nagy                           Helvetius
    13 Accorti                        Paolo
    13 O'Rourke                       Timothy
    13 Schmitt                        Carine
    14 Afonso                         Pedro
```

```
    14 Josephs                     Karin
    14 Lincoln                     Elizabeth
(43 rows affected)
```

The sort on first names in this query doesn't affect the result set because no two employees have the same last name and the same job ID.

Now let's take a look at an ORDER BY clause with a GROUP BY clause and an aggregate function:

```
SELECT   type, pub_id, AVG(price) AS "Average Price"
FROM     titles
GROUP BY type, pub_id
ORDER BY type
```

The result set looks like this:

```
type         pub_id Average Price
———————————— ————— ————————————————————
UNDECIDED    0877                    NULL
business     0736                    2.99
business     1389                   17.31
mod_cook     0877                   11.49
popular_comp 1389                   21.48
psychology   0736                   11.48
psychology   0877                   21.59
trad_cook    0877                   15.96
(8 rows affected)
```

The results are sorted in alphabetical order (ascending) by type. Also, notice that in this query, both *type* and *pub_id* must be in the GROUP BY clause because they are not part of an aggregate function. If you had left the *pub_id* column out of the GROUP BY clause, SQL Server would have displayed an error message.

You cannot use aggregate functions or subqueries in the ORDER BY clause. However, if you had given an alias to an aggregate in the SELECT clause, you could use it in the ORDER BY clause, as shown here:

```
SELECT   type, pub_id, AVG(price) AS "Average Price"
FROM     titles
GROUP BY type, pub_id
ORDER BY "Average Price"
```

The result set looks like this:

```
type         pub_id Average Price
———————————— ————— ———————————————————
UNDECIDED    0877                  NULL
business     0736                  2.99
psychology   0736                 11.48
mod_cook     0877                 11.49
psychology   0877                 21.59
trad_cook    0877                 15.96
business     1389                 17.31
popular_comp 1389                 21.48
(8 rows affected)
```

Now the results are ordered by average price. *NULL* is considered lowest in the sort order, so it is at the top of the list.