Chapter 6

Queries

In This Chapter

▶ SELECT Statement: Its Clauses and Functions
▶ Subqueries
▶ Temporary Tables
▶ Join Operator
▶ Correlated Subqueries
▶ Table Expressions
In this chapter you will learn how to use the SELECT statement to perform retrievals. This chapter describes every clause in this statement and gives numerous examples using the sample database to demonstrate the practical use of each clause. After that, the chapter introduces aggregate functions and the set operators, as well as computed columns and temporary tables. The second part of the chapter tells you more about complex queries. It introduces the join operator, which is the most important operator for relational database systems, and looks at all its forms. Correlated subqueries and the EXISTS function are then introduced. The end of the chapter describes common table expressions, together with the APPLY operator.

SELECT Statement: Its Clauses and Functions

The Transact-SQL language has one basic statement for retrieving information from a database: the SELECT statement. With this statement, it is possible to query information from one or more tables of a database (or even from multiple databases). The result of a SELECT statement is another table, also known as a result set.

The simplest form of the SELECT statement contains a SELECT list with the FROM clause. (All other clauses are optional.) This form of the SELECT statement has the following syntax:

```
SELECT [ ALL | DISTINCT] column_list
FROM {table1 [tab_alias1] } ,...
```

- **table1** is the name of the table from which information is retrieved. **tab_alias1** provides an alias for the name of the corresponding table. An alias is another name for the corresponding table and can be used as a shorthand way to refer to the table or as a way to refer to two logical instances of the same physical table. Don’t worry; this will become clearer as examples are presented.
- **column_list** contains one or more of the following specifications:

  - The asterisk symbol (*), which specifies all columns of the named tables in the FROM clause (or from a single table when qualified, as in table2.*)
  - The explicit specification of column names to be retrieved
  - The specification **column_name [AS] column_heading**, which is a way to replace the name of a column or to assign a new name to an expression
  - An expression
  - A system or an aggregate function
NOTE
In addition to the preceding specifications, there are other options that will be presented later in this chapter.

A SELECT statement can retrieve either columns or rows from a table. The first operation is called *SELECT list* (or *projection*), and the second one is called *selection*. The combination of both operations is also possible in a SELECT statement.

NOTE
Before you start to execute queries in this chapter, re-create the entire *sample* database.

Example 6.1 shows the simplest retrieval form with the SELECT statement.

**EXAMPLE 6.1**
Get full details of all departments:

```
USE sample;
SELECT dept_no, dept_name, location
FROM department;
```

The result is

<table>
<thead>
<tr>
<th>dept_no</th>
<th>dept_name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
</tbody>
</table>

The SELECT statement in Example 6.1 retrieves all rows and all columns from the *department* table. If you include all columns of a table in a SELECT list (as in Example 6.1), you can use `*` as shorthand, but this notation is not recommended. The column names serve as column headings of the resulting output.

The simplest form of the SELECT statement just described is not very useful for queries. In practice, there are always several more clauses in a SELECT statement than...
in the statement shown in Example 6.1. The following is the syntax of a SELECT statement that references a table, with (almost) all possible clauses included:

```
SELECT select_list
    [INTO new_table_]
FROM table
    [WHERE search_condition]
    [GROUP BY group_by_expression]
    [HAVING search_condition]
    [ORDER BY order_expression [ASC | DESC ] ];
```

**NOTE**

The clauses in the SELECT statement must be written in the syntactical order given in the preceding syntax—for example, the GROUP BY clause must come after the WHERE clause and before the HAVING clause. However, because the INTO clause is not as significant as the other clauses, it will be discussed later in the chapter, after the other clauses have been discussed.

The following subsections describe the clauses that can be used in a query, WHERE, GROUP BY, HAVING, and ORDER BY, as well as aggregate functions, the IDENTITY property, the new sequences feature, set operators, and the CASE expression.

### WHERE Clause

Often, it is necessary to define one or more conditions that limit the selected rows. The WHERE clause specifies a Boolean expression (an expression that returns a value of TRUE or FALSE) that is tested for each row to be returned (potentially). If the expression is true, then the row is returned; if it is false, it is discarded.

Example 6.2 shows the use of the WHERE clause.

**EXAMPLE 6.2**

Get the names and numbers of all departments located in Dallas:

```
USE sample;
SELECT dept_name, dept_no
    FROM department
    WHERE location = 'Dallas';
```

The result is

<table>
<thead>
<tr>
<th>dept_name</th>
<th>dept_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research</td>
<td>d1</td>
</tr>
<tr>
<td>Marketing</td>
<td>d3</td>
</tr>
</tbody>
</table>
In addition to the equal sign, the WHERE clause can contain other comparison operators, including the following:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&gt; (or !=)</td>
<td>not equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>!&gt;</td>
<td>not greater than</td>
</tr>
<tr>
<td>!&lt;</td>
<td>not less than</td>
</tr>
</tbody>
</table>

Example 6.3 shows the use of a comparison operator in the WHERE clause.

**EXAMPLE 6.3**
Get the last and first names of all employees with employee numbers greater than or equal to 15000:

```
USE sample;
SELECT emp_lname, emp_fname
FROM employee
WHERE emp_no >= 15000;
```

The result is

<table>
<thead>
<tr>
<th>emp_lname</th>
<th>emp_fname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Matthew</td>
</tr>
<tr>
<td>Barrimore</td>
<td>John</td>
</tr>
<tr>
<td>James</td>
<td>James</td>
</tr>
<tr>
<td>Moser</td>
<td>Sybill</td>
</tr>
</tbody>
</table>

An expression can also be a part of the condition in the WHERE clause, as Example 6.4 shows.

**EXAMPLE 6.4**
Get the project names for all projects with a budget > 60000 £. The current rate of exchange is 0.51 £ per $1.

```
USE sample;
SELECT project_name
FROM project
WHERE budget*0.51 > 60000;
```
The result is

<table>
<thead>
<tr>
<th>project_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apollo</td>
</tr>
<tr>
<td>Mercury</td>
</tr>
</tbody>
</table>

Comparisons of strings (that is, values of data types CHAR, VARCHAR, NCHAR, or NVARCHAR) are executed in accordance with the collating sequence in effect (the “sort order” specified when the Database Engine was installed). If two strings are compared using ASCII code (or any other code), each of the corresponding (first, second, third, and so on) characters will be compared. One character is lower in priority than the other if it appears in the code table before the other one. Two strings of different lengths are compared after the shorter one is padded at the right with blanks, so that the length of both strings is equal. Numbers compare algebraically. Values of temporal data types (such as DATE, TIME, and DATETIME) compare in chronological order.

**Boolean Operators**

WHERE clause conditions can either be simple or contain multiple conditions. Multiple conditions can be built using the Boolean operators AND, OR, and NOT. The behavior of these operators was described in Chapter 4 using truth tables.

If two conditions are connected by the AND operator, rows are retrieved for which both conditions are true. If two conditions are connected by the OR operator, all rows of a table are retrieved in which either the first or the second condition (or both) is true, as shown in Example 6.5.

**EXAMPLE 6.5**

Get the employee numbers for all employees who work for either project p1 or project p2 (or both):

```sql
USE sample;
SELECT project_no, emp_no
FROM works_on
WHERE project_no = 'p1'
  OR project_no = 'p2';
```
The result is

<table>
<thead>
<tr>
<th>project_no</th>
<th>emp_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>10102</td>
</tr>
<tr>
<td>p2</td>
<td>25348</td>
</tr>
<tr>
<td>p2</td>
<td>18316</td>
</tr>
<tr>
<td>p2</td>
<td>29346</td>
</tr>
<tr>
<td>p1</td>
<td>9031</td>
</tr>
<tr>
<td>p1</td>
<td>28559</td>
</tr>
<tr>
<td>p2</td>
<td>28559</td>
</tr>
<tr>
<td>p1</td>
<td>29346</td>
</tr>
</tbody>
</table>

The result of Example 6.5 contains some duplicate values of the `emp_no` column. To eliminate this redundant information, use the DISTINCT option, as shown here:

```sql
USE sample;
SELECT DISTINCT emp_no
FROM works_on
WHERE project_no = 'p1'
OR project_no = 'p2';
```

In this case, the result is

<table>
<thead>
<tr>
<th>emp_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>9031</td>
</tr>
<tr>
<td>10102</td>
</tr>
<tr>
<td>18316</td>
</tr>
<tr>
<td>25348</td>
</tr>
<tr>
<td>28559</td>
</tr>
<tr>
<td>29346</td>
</tr>
</tbody>
</table>

Note that the DISTINCT option can be used only once in a SELECT list, and it must precede all column names in that list. Therefore, Example 6.6 is wrong.

**EXAMPLE 6.6 (EXAMPLE OF AN ILLEGAL STATEMENT)**

```sql
USE sample;
SELECT emp_fname, DISTINCT emp_no
```
FROM employee
WHERE emp_lname = 'Moser';

The result is

Server: Msg 156, Level 15, State 1, Line 1
Incorrect syntax near the keyword 'DISTINCT'.

**NOTE**

*When there is more than one column in the SELECT list, the DISTINCT clause displays all rows where the combination of columns is distinct.*

The WHERE clause may include any number of the same or different Boolean operations. You should be aware that the three Boolean operations have different priorities for evaluation: the NOT operation has the highest priority, AND is evaluated next, and the OR operation has the lowest priority. If you do not pay attention to these different priorities for Boolean operations, you will get unexpected results, as Example 6.7 shows.

**EXAMPLE 6.7**

USE sample;
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE emp_no = 25348 AND emp_lname = 'Smith'
OR emp_fname = 'Matthew' AND dept_no = 'd1';

SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE ((emp_no = 25348 AND emp_lname = 'Smith')
OR emp_fname = 'Matthew') AND dept_no = 'd1';

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
</tr>
</tbody>
</table>

As the results of Example 6.7 show, the two SELECT statements display two different result sets. In the first SELECT statement, the system evaluates both AND operators first (from the left to the right), and then evaluates the OR operator. In the
second SELECT statement, the use of parentheses changes the operation execution, with all expressions within parentheses being executed first, in sequence from left to right. As you can see, the first statement returned one row, while the second statement returned zero rows.

The existence of several Boolean operations in a WHERE clause complicates the corresponding SELECT statement and makes it error prone. In such cases, the use of parentheses is highly recommended, even if they are not necessary. The readability of such SELECT statements will be greatly improved, and possible errors can be avoided. Here is the first SELECT statement from Example 6.7, modified using the recommended form:

```
USE sample;
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE (emp_no = 25348 AND emp_lname = 'Smith')
OR (emp_fname = 'Matthew' AND dept_no = 'd1');
```

The third Boolean operator, NOT, changes the logical value of the corresponding condition. The truth table for NOT in Chapter 4 shows that the negation of the TRUE value is FALSE and vice versa; the negation of the NULL value is also NULL.

Example 6.8 shows the use of the NOT operator.

**EXAMPLE 6.8**

Get the employee numbers and first names of all employees who do not belong to the department d2:

```
USE sample
SELECT emp_no, emp_lname
FROM employee
WHERE NOT dept_no = 'd2';
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Smith</td>
</tr>
<tr>
<td>10102</td>
<td>Jones</td>
</tr>
<tr>
<td>18316</td>
<td>Barrimore</td>
</tr>
<tr>
<td>28559</td>
<td>Moser</td>
</tr>
</tbody>
</table>

In this case, the NOT operator can be replaced by the comparison operator <> (not equal).
NOTE
This book uses the operator <> (instead of !=) to remain consistent with the ANSI SQL standard.

IN and BETWEEN Operators
An IN operator allows the specification of two or more expressions to be used for a query search. The result of the condition returns TRUE if the value of the corresponding column equals one of the expressions specified by the IN predicate.

Example 6.9 shows the use of the IN operator.

EXAMPLE 6.9
Get all the columns for every employee whose employee number equals 29346, 28559, or 25348:

USE sample;
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE emp_no IN (29346, 28559, 25348);

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
</tr>
</tbody>
</table>

An IN operator is equivalent to a series of conditions, connected with one or more OR operators. (The number of OR operators is equal to the number of expressions following the IN operator minus one.)

The IN operator can be used together with the Boolean operator NOT, as shown in Example 6.10. In this case, the query retrieves rows that do not include any of the listed values in the corresponding columns.

EXAMPLE 6.10
Get all columns for every employee whose employee number is neither 10102 nor 9031:

USE sample;
SELECT emp_no, emp_fname, emp_lname, dept_no
FROM employee
WHERE emp_no NOT IN (10102, 9031);

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
<th>dept_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>d3</td>
</tr>
<tr>
<td>18316</td>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
<td>d2</td>
</tr>
<tr>
<td>2581</td>
<td>Elke</td>
<td>Hansel</td>
<td>d2</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>d1</td>
</tr>
</tbody>
</table>

In contrast to the IN operator, which specifies each individual value, the BETWEEN operator specifies a range, which determines the lower and upper bounds of qualifying values. Example 6.11 provides an example.

**EXAMPLE 6.11**

Get the names and budgets for all projects with a budget between $95,000 and $120,000, inclusive:

```sql
USE sample;
SELECT project_name, budget
FROM project
WHERE budget BETWEEN 95000 AND 120000;
```

The result is

<table>
<thead>
<tr>
<th>project_name</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apollo</td>
<td>120000</td>
</tr>
<tr>
<td>Gemini</td>
<td>95000</td>
</tr>
</tbody>
</table>

The BETWEEN operator searches for all values in the range inclusively; that is, qualifying values can be between or equal to the lower and upper boundary values.

The BETWEEN operator is logically equal to two individual comparisons, which are connected with the Boolean operator AND. Example 6.11 is equivalent to Example 6.12.

**EXAMPLE 6.12**

```sql
USE sample;
SELECT project_name, budget
```
Microsoft SQL Server 2012: A Beginner’s Guide

FROM project
WHERE budget >= 95000 AND budget <= 120000;

Like the BETWEEN operator, the NOT BETWEEN operator can be used to search for column values that do not fall within the specified range. The BETWEEN operator can also be applied to columns with character and date values.

The two SELECT statements in Example 6.13 show a query that can be written in two different, but equivalent, ways.

**EXAMPLE 6.13**

Get the names of all projects with a budget less than $100,000 and greater than $150,000:

USE sample;
SELECT project_name
FROM project
WHERE budget NOT BETWEEN 100000 AND 150000;

The result is

<table>
<thead>
<tr>
<th>project_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gemini</td>
</tr>
<tr>
<td>Mercury</td>
</tr>
</tbody>
</table>

Using comparison operators, the query looks different:

USE sample;
SELECT project_name
FROM project
WHERE budget < 100000 OR budget > 150000;

**NOTE**

Although the English phrasing of the requirements, “Get the names of all projects with budgets that are less than $100,000 and greater than $150,000,” suggests the use of the AND operator in the second SELECT statement presented in Example 6.13, the logical meaning of the query demands the use of the OR operator, because if you use AND instead of OR, you will get no results at all. (The reason is that there cannot be a budget that is at the same time less than $100,000 and greater than $150,000.) Therefore, the second query in the example shows a possible problem that can appear between English phrasing of an exercise and its logical meaning.
Queries Involving NULL Values

A NULL in the CREATE TABLE statement specifies that a special value called NULL (which usually represents unknown or not applicable values) is allowed in the column. These values differ from all other values in a database. The WHERE clause of a SELECT statement generally returns rows for which the comparison evaluates to TRUE. The concern, then, regarding queries is, how will comparisons involving NULL values be evaluated in the WHERE clause?

All comparisons with NULL values will return FALSE (even when preceded by NOT). To retrieve the rows with NULL values in the column, Transact-SQL includes the operator feature IS NULL. This specification in a WHERE clause of a SELECT statement has the following general form:

column IS [NOT] NULL

Example 6.14 shows the use of the IS NULL operator.

EXAMPLE 6.14
Get employee numbers and corresponding project numbers for employees with unknown jobs who work on project p2:

USE sample;
SELECT emp_no, project_no
FROM works_on
WHERE project_no = 'p2'
AND job IS NULL;

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>project_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>18316</td>
<td>p2</td>
</tr>
<tr>
<td>29346</td>
<td>p2</td>
</tr>
</tbody>
</table>

Because all comparisons with NULL values return FALSE, Example 6.15 shows syntactically correct, but logically incorrect, usage of NULL.

EXAMPLE 6.15
USE sample;
SELECT project_no, job
FROM works_on
WHERE job <> NULL;
The result is

<table>
<thead>
<tr>
<th>project_no</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The condition “column IS NOT NULL” is equivalent to the condition “NOT (column IS NULL).”

The system function ISNULL allows a display of the specified value as substitution for NULL (see Example 6.16).

**EXAMPLE 6.16**

```
USE sample;
SELECT emp_no, ISNULL(job, 'Job unknown') AS task
FROM works_on
WHERE project_no = 'p1';
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>task</th>
</tr>
</thead>
<tbody>
<tr>
<td>10102</td>
<td>Analyst</td>
</tr>
<tr>
<td>9031</td>
<td>Manager</td>
</tr>
<tr>
<td>28559</td>
<td>Job unknown</td>
</tr>
<tr>
<td>29346</td>
<td>Clerk</td>
</tr>
</tbody>
</table>

Example 6.16 uses a column heading called **task** for the **job** column.

**LIKE Operator**

LIKE is an operator that is used for pattern matching; that is, it compares column values with a specified pattern. The data type of the column can be any character or date. The general form of the LIKE operator is

`column [NOT] LIKE 'pattern'`

*pattern* may be a string or date constant or expression (including columns of tables) and must be compatible with the data type of the corresponding column. For the specified column, the comparison between the value in a row and the pattern evaluates to TRUE if the column value matches the pattern expression.

Certain characters within the pattern—called wildcard characters—have a specific interpretation. Two of them are

- `%` (percent sign)  Specifies any sequence of zero or more characters
- `_` (underscore)   Specifies any single character
Example 6.17 shows the use of the wildcard characters % and _.

**EXAMPLE 6.17**

Get the first and last names and numbers of all employees whose first name contains the letter \( a \) as the second character:

```sql
USE sample;
SELECT emp_fname, emp_lname, emp_no
FROM employee
WHERE emp_fname LIKE '_a%';
```

The result is

<table>
<thead>
<tr>
<th>emp_fname</th>
<th>emp_lname</th>
<th>emp_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matthew</td>
<td>Smith</td>
<td>25348</td>
</tr>
<tr>
<td>James</td>
<td>James</td>
<td>29346</td>
</tr>
</tbody>
</table>

In addition to the percent sign and the underscore, Transact-SQL supports other characters that have a special meaning when used with the LIKE operator. These characters ([ ], and ^) are demonstrated in Examples 6.18 and 6.19.

**EXAMPLE 6.18**

Get full details of all departments whose location begins with a character in the range \( C \) through \( F \):

```sql
USE sample;
SELECT  dept_nt, dept_name, location
FROM department
WHERE location LIKE '[C-F]%';
```

The result is

<table>
<thead>
<tr>
<th>dept_no</th>
<th>dept_name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
</tbody>
</table>

As shown in Example 6.18, the square brackets, [ ], delimit a range or list of characters. The order in which characters appear in a range is defined by the collating sequence, which is determined during the system installation.

The character ^ specifies the negation of a range or a list of characters. This character has this meaning only within a pair of square brackets, as shown in Example 6.19.
**EXAMPLE 6.19**

Get the numbers and first and last names of all employees whose last name does not begin with the letter \( J, K, L, M, N, \) or \( O \) and whose first name does not begin with the letter \( E \) or \( Z \):

```sql
USE sample;
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE emp_lname LIKE '[^J-O]%' AND emp_fname LIKE '[^EZ]';
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
</tr>
<tr>
<td>18316</td>
<td>John</td>
<td>Barrimore</td>
</tr>
</tbody>
</table>

The condition “column NOT LIKE ‘pattern’” is equivalent to the condition “NOT (column LIKE ‘pattern’).” Example 6.20 shows the use of the LIKE operator (together with NOT).

**EXAMPLE 6.20**

Get full details of all employees whose first name does not end with the character \( n \):

```sql
USE sample;
SELECT emp_no, emp_fname, emp_lname
FROM employee
WHERE emp_fname NOT LIKE '%n';
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
</tr>
<tr>
<td>2581</td>
<td>Elke</td>
<td>Hansel</td>
</tr>
<tr>
<td>9031</td>
<td>Elsa</td>
<td>Berton</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
</tr>
</tbody>
</table>

Any of the wildcard characters (%, _, [ , ] , or ^) enclosed in square brackets stands for itself. An equivalent feature is available through the ESCAPE option. Therefore, both SELECT statements in Example 6.21 have the same meaning.
EXAMPLE 6.21

USE sample;
SELECT project_no, project_name
FROM project
  WHERE project_name LIKE '%[_]%' ;
SELECT project_no, project_name
FROM project
  WHERE project_name LIKE '%!_%' ESCAPE '!';

The result is

<table>
<thead>
<tr>
<th>project_no</th>
<th>project_name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both SELECT statements search for the underscore as an actual character in the column project_name. In the first SELECT statement, this search is established by enclosing the sign _ in square brackets. The second SELECT statement uses a character (in Example 6.21, the character !) as an escape character. The escape character overrides the meaning of the underscore as the wildcard character and leaves it to be interpreted as an ordinary character. (The result contains no rows because there are no project names that include the underscore character.)

NOTE
The SQL standard supports the use of only %, _, and the ESCAPE operator. For this reason, if any wildcard character must stand for itself, using the ESCAPE operator instead of a pair of square brackets is recommended.

GROUP BY Clause
The GROUP BY clause defines one or more columns as a group such that all rows within any group have the same values for those columns. Example 6.22 shows the simple use of the GROUP BY clause.

EXAMPLE 6.22
Get all jobs of the employees:

USE sample;
SELECT job
  FROM works_on
  GROUP BY job;
The result is

<table>
<thead>
<tr>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
</tr>
<tr>
<td>Analyst</td>
</tr>
<tr>
<td>Clerk</td>
</tr>
<tr>
<td>Manager</td>
</tr>
</tbody>
</table>

In Example 6.22, the GROUP BY clause builds different groups for all possible values (NULL, too) appearing in the job column.

**NOTE**

There is a restriction regarding the use of columns in the GROUP BY clause. Each column appearing in the SELECT list of the query must also appear in the GROUP BY clause. This restriction does not apply to constants and to columns that are part of an aggregate function. (Aggregate functions are explained in the next subsection.) This makes sense, because only columns in the GROUP BY clause are guaranteed to have a single value for each group.

A table can be grouped by any combination of its columns. Example 6.23 shows the grouping of rows of the works_on table using two columns.

**EXAMPLE 6.23**

Group all employees using their project numbers and jobs:

```sql
USE sample;
SELECT project_no, job
FROM works_on
GROUP BY project_no, job;
```

The result is

<table>
<thead>
<tr>
<th>project_no</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>Analyst</td>
</tr>
<tr>
<td>p1</td>
<td>Clerk</td>
</tr>
<tr>
<td>p1</td>
<td>Manager</td>
</tr>
<tr>
<td>p2</td>
<td>NULL</td>
</tr>
<tr>
<td>p2</td>
<td>Clerk</td>
</tr>
<tr>
<td>p3</td>
<td>Analyst</td>
</tr>
<tr>
<td>p3</td>
<td>Clerk</td>
</tr>
<tr>
<td>p3</td>
<td>Manager</td>
</tr>
</tbody>
</table>
The result of Example 6.23 shows that there are nine groups with different combinations of project numbers and jobs. The only two groups that contain more than one row are

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>Clerk</td>
<td>25348, 28559</td>
</tr>
<tr>
<td>p2</td>
<td>NULL</td>
<td>18316, 29346</td>
</tr>
</tbody>
</table>

The sequence of the column names in the GROUP BY clause does not need to correspond to the sequence of the names in the SELECT list.

### Aggregate Functions

Aggregate functions are functions that are used to get summary values. All aggregate functions can be divided into several groups:

- **Convenient aggregate functions**
- **Statistical aggregate functions**
- **User-defined aggregate functions**
- **Analytic aggregate functions**

The first three types are described in the following sections, while analytic aggregate functions are explained in detail in Chapter 23.

### Convenient Aggregate Functions

The Transact-SQL language supports six aggregate functions:

- **MIN**
- **MAX**
- **SUM**
- **AVG**
- **COUNT**
- **COUNT_BIG**

All aggregate functions operate on a single argument, which can be either a column or an expression. (The only exception is the second form of the COUNT and COUNT_BIG functions, COUNT(*) and COUNT_BIG(*).) The result of each aggregate function is a constant value, which is displayed in a separate column of the result.
The aggregate functions appear in the SELECT list, which can include a GROUP BY clause. If there is no GROUP BY clause in the SELECT statement, and the SELECT list includes at least one aggregate function, then no simple columns can be included in the SELECT list (other than as arguments of an aggregate function). Therefore, Example 6.24 is wrong.

**EXAMPLE 6.24 (EXAMPLE OF AN ILLEGAL STATEMENT)**

```
USE sample;
SELECT emp_lname, MIN(emp_no)
     FROM employee;
```

The `emp_lname` column of the `employee` table must not appear in the SELECT list of Example 6.24 because it is not the argument of an aggregate function. On the other hand, all column names that are not arguments of an aggregate function may appear in the SELECT list if they are used for grouping.

The argument of an aggregate function can be preceded by one of two keywords:

- **ALL** Indicates that all values of a column are to be considered (ALL is the default value)
- **DISTINCT** Eliminates duplicate values of a column before the aggregate function is applied

**MIN and MAX Aggregate Functions** The aggregate functions MIN and MAX compute the lowest and highest values in the column, respectively. If there is a WHERE clause, the MIN and MAX functions return the lowest or highest of values from selected rows. Example 6.25 shows the use of the aggregate function MIN.

**EXAMPLE 6.25**

Get the lowest employee number:

```
USE sample;
SELECT MIN(emp_no) AS min_employee_no
     FROM employee;
```

The result is

```
min_employee_no
2581
```

The result of Example 6.25 is not user friendly. For instance, the name of the employee with the lowest number is not known. As already shown, the explicit
specification of the emp_name column in the SELECT list is not allowed. To retrieve the name of the employee with the lowest employee number, use a subquery, as shown in Example 6.26, where the inner query contains the SELECT statement of the previous example.

**EXAMPLE 6.26**

Get the number and the last name of the employee with the lowest employee number:

```
USE sample;
SELECT emp_no, emp_lname
FROM employee
WHERE emp_no =
    (SELECT MIN(emp_no)
     FROM employee);
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>2581</td>
<td>Hansel</td>
</tr>
</tbody>
</table>

Example 6.27 shows the use of the aggregate function MAX.

**EXAMPLE 6.27**

Get the employee number of the manager who was entered last in the works_on table:

```
USE sample;
SELECT emp_no
FROM works_on
WHERE enter_date =
    (SELECT MAX(enter_date)
     FROM works_on
     WHERE job = 'Manager');
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>10102</td>
</tr>
</tbody>
</table>

The argument of the functions MIN and MAX can also be a string value or a date. If the argument has a string value, the comparison between all values will be provided
using the actual collating sequence. For all arguments of temporal data types, the earliest date specifies the lowest value in the column and the latest date specifies the highest value in the column.

The DISTINCT option cannot be used with the aggregate functions MIN and MAX. All NULL values in the column that are the argument of the aggregate function MIN or MAX are always eliminated before MIN or MAX is applied.

**SUM Aggregate Function** The aggregate function SUM calculates the sum of the values in the column. The argument of the function SUM must be numeric. Example 6.28 shows the use of the SUM function.

**EXAMPLE 6.28**

Calculate the sum of all budgets of all projects:

```sql
USE sample;
SELECT SUM(budget) sum_of_budgets
FROM project;
```

The result is

<table>
<thead>
<tr>
<th>sum_of_budgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>401500</td>
</tr>
</tbody>
</table>

The aggregate function in Example 6.28 groups all values of the projects’ budgets and determines their total sum. For this reason, the query in Example 6.28 (as does each analog query) implicitly contains the grouping function. The grouping function from Example 6.28 can be written explicitly in the query, as shown in Example 6.29.

**EXAMPLE 6.29**

```sql
SELECT SUM(budget) sum_of_budgets
FROM project
GROUP BY();
```

The use of this syntax for the GROUP BY clause is recommended because it defines a grouping explicitly. (Chapter 23 describes several other GROUP BY features.)

The use of the DISTINCT option eliminates all duplicate values in the column before the function SUM is applied. Similarly, all NULL values are always eliminated before SUM is applied.

**AVG Aggregate Function** The aggregate function AVG calculates the average of the values in the column. The argument of the function AVG must be numeric. All NULL
values are eliminated before the function AVG is applied. Example 6.30 shows the use of the AVG aggregate function.

**EXAMPLE 6.30**

Calculate the average of all budgets with an amount greater than $100,000:

```sql
USE sample;
SELECT AVG(budget) avg_budget
FROM project
WHERE budget > 100000;
```

The result is

<table>
<thead>
<tr>
<th>avg_budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>153250</td>
</tr>
</tbody>
</table>

**COUNT and COUNT_BIG Aggregate Functions**

The aggregate function COUNT has two different forms:

- `COUNT([DISTINCT] col_name)`
- `COUNT(*)`

The first form calculates the number of values in the `col_name` column. When the `DISTINCT` keyword is used, all duplicate values are eliminated before COUNT is applied. This form of COUNT does not count NULL values for the column.

Example 6.31 shows the use of the first form of the aggregate function COUNT.

**EXAMPLE 6.31**

Count all different jobs in each project:

```sql
USE sample;
SELECT project_no, COUNT(DISTINCT job) job_count
FROM works_on
GROUP BY project_no;
```

The result is

<table>
<thead>
<tr>
<th>project_no</th>
<th>job_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>3</td>
</tr>
<tr>
<td>p2</td>
<td>1</td>
</tr>
<tr>
<td>p3</td>
<td>3</td>
</tr>
</tbody>
</table>
As can be seen from the result of Example 6.31, all NULL values are eliminated before the function \( \text{COUNT(DISTINCT job)} \) is applied. (The sum of all values in the column is 8 instead of 11.)

The second form of the function \( \text{COUNT, COUNT(*)} \), counts the number of rows in the table. Or, if there is a WHERE clause in the SELECT statement, \( \text{COUNT(*)} \) returns the number of rows for which the WHERE condition is true. In contrast to the first form of the function \( \text{COUNT} \), the second form does not eliminate NULL values, because this function operates on rows and not on columns. Example 6.32 shows the use of \( \text{COUNT(*)} \).

**EXAMPLE 6.32**

Get the number of each job in all projects:

```sql
USE sample;
SELECT job, COUNT(*) job_count
FROM works_on
GROUP BY job;
```

The result is

<table>
<thead>
<tr>
<th>job</th>
<th>job_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>3</td>
</tr>
<tr>
<td>Analyst</td>
<td>2</td>
</tr>
<tr>
<td>Clerk</td>
<td>4</td>
</tr>
<tr>
<td>Manager</td>
<td>2</td>
</tr>
</tbody>
</table>

The COUNT_BIG function is analogous to the COUNT function. The only difference between them is in relation to their return values: COUNT_BIG always returns a value of the BIGINT data type, while the COUNT function always returns a value of the INTEGER data type.

**Statistical Aggregate Functions**

The following aggregate functions belong to the group of statistical aggregate functions:

- **VAR** Computes the variance of all the values listed in a column or expression
- **VARP** Computes the variance for the population of all the values listed in a column or expression
Chapter 6: Queries

- **STDEV** Computes the standard deviation (which is computed as the square root of the corresponding variance) of all the values listed in a column or expression

- **STDEVP** Computes the standard deviation for the population of all the values listed in a column or expression

Examples showing statistical aggregate functions will be provided in Chapter 23.

**User-Defined Aggregate Functions**

The Database Engine also supports the implementation of user-defined aggregate functions. Using these functions, you can implement and deploy aggregate functions that do not belong to aggregate functions supported by the system. These functions are a special case of user-defined functions, which will be described in detail in Chapter 8.

**HAVING Clause**

The HAVING clause defines the condition that is then applied to groups of rows. Hence, this clause has the same meaning to groups of rows that the WHERE clause has to the content of the corresponding table. The syntax of the HAVING clause is

```
HAVING condition
```

where **condition** contains aggregate functions or constants.

Example 6.33 shows the use of the HAVING clause with the aggregate function **COUNT(*)**.

**EXAMPLE 6.33**

Get project numbers for all projects employing fewer than four persons:

```
USE sample;
SELECT project_no
FROM works_on
GROUP BY project_no
HAVING COUNT(*) < 4;
```

The result is

<table>
<thead>
<tr>
<th>project_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>p3</td>
</tr>
</tbody>
</table>
In Example 6.33, the system uses the GROUP BY clause to group all rows according to existing values in the `project_no` column. After that, it counts the number of rows in each group and selects those groups with three or fewer rows.

The HAVING clause can also be used without aggregate functions, as shown in Example 6.34.

**EXAMPLE 6.34**

Group rows of the `works_on` table by job and eliminate those jobs that do not begin with the letter M:

```sql
USE sample;
SELECT job
FROM works_on
GROUP BY job
HAVING job LIKE 'M%';
```

The result is

<table>
<thead>
<tr>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manager</td>
</tr>
</tbody>
</table>

The HAVING clause can also be used without the GROUP BY clause, although doing so is uncommon in practice. In such a case, all rows of the entire table belong to a single group.

**ORDER BY Clause**

The ORDER BY clause defines the particular order of the rows in the result of a query. This clause has the following syntax:

```sql
ORDER BY { [col_name | col_number [ASC | DESC] ] , ... }
```

The `col_name` column defines the order. `col_number` is an alternative specification that identifies the column by its ordinal position in the sequence of all columns in the SELECT list (1 for the first column, 2 for the second one, and so on). ASC indicates ascending order and DESC indicates descending order, with ASC as the default value.

**NOTE**

The columns in the ORDER BY clause need not appear in the SELECT list. However, the ORDER BY columns must appear in the SELECT list if SELECT DISTINCT is specified. Also, this clause cannot reference columns from tables that are not listed in the FROM clause.
As the syntax of the ORDER BY clause shows, the order criterion may contain more than one column, as shown in Example 6.35.

**EXAMPLE 6.35**

Get department numbers and employee names for employees with employee numbers < 20000, in ascending order of last and first names:

```sql
USE sample;
SELECT emp_fname, emp_lname, dept_no
FROM employee
WHERE emp_no < 20000
ORDER BY emp_lname, emp_fname;
```

The result is

<table>
<thead>
<tr>
<th>emp_fname</th>
<th>emp_lname</th>
<th>dept_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
</tr>
<tr>
<td>Elsa</td>
<td>Bertoni</td>
<td>d2</td>
</tr>
<tr>
<td>Elke</td>
<td>Hansel</td>
<td>d2</td>
</tr>
<tr>
<td>Ann</td>
<td>Jones</td>
<td>d3</td>
</tr>
</tbody>
</table>

It is also possible to identify the columns in the ORDER BY clause by the ordinal position of the column in the SELECT list. The ORDER BY clause in Example 6.35 could be written in the following form:

```sql
ORDER BY 2,1
```

The use of column numbers instead of column names is an alternative solution if the order criterion contains any aggregate function. (The other way is to use column headings, which then appear in the ORDER BY clause.) However, using column names rather than numbers in the ORDER BY clause is recommended, to reduce the difficulty of maintaining the query if any columns need to be added or deleted from the SELECT list. Example 6.36 shows the use of column numbers.

**EXAMPLE 6.36**

For each project number, get the project number and the number of all employees, in descending order of the employee number:

```sql
USE sample;
SELECT project_no, COUNT(*) emp_quantity
```
FROM works_on
GROUP BY project_no
ORDER BY 2 DESC

The result is

<table>
<thead>
<tr>
<th>project_no</th>
<th>emp_quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>4</td>
</tr>
<tr>
<td>p2</td>
<td>4</td>
</tr>
<tr>
<td>p3</td>
<td>3</td>
</tr>
</tbody>
</table>

The Transact-SQL language orders NULL values at the beginning of all values if the order is ascending and orders them at the end of all values if the order is descending.

**Using ORDER BY to Support Paging**

If you want to display rows on the current page, you can either implement that in your application or instruct the database server to do it. In the former case, all rows from the database are sent to the application, and the application's task is to retrieve the rows needed for printing and to display them. In the latter case, only those rows needed for the current page are selected from the server side and displayed. As you might guess, server-side paging generally provides better performance, because only the rows needed for printing are sent to the client.

SQL Server 2012 introduces two new clauses of the SELECT statement, OFFSET and FETCH, to support server-side paging. Example 6.37 shows the use of these two clauses.

**EXAMPLE 6.37**

Get the business entity ID, job title, and birthday for all female employees from the AdventureWorks database in ascending order of job title. Display the third page. (Ten rows are displayed per page.)

USE AdventureWorks;
SELECT BusinessEntityID, JobTitle, BirthDate
FROM HumanResources.Employee
WHERE Gender = 'F'
ORDER BY JobTitle
OFFSET 20 ROWS
FETCH NEXT 10 ROWS ONLY;
NOTE
You can find further examples concerning the OFFSET clause in Chapter 23 (see Examples 23.24 and 23.25).

The OFFSET clause specifies the number of rows to skip before starting to return the rows. This is evaluated after the ORDER BY clause is evaluated and the rows are sorted. The FETCH NEXT clause specifies the number of rows to retrieve. The parameter of this clause can be a constant, an expression, or a result of a query. FETCH NEXT is analogous to FETCH FIRST.

The main purpose of server-side paging is to implement generic page forms, using variables. This can be done using a SQL Server batch. The corresponding example can be found in Chapter 8 (see Example 8.5).

SELECT Statement and IDENTITY Property
The IDENTITY property allows you to specify a counter of values for a specific column of a table. Columns with numeric data types, such as TINYINT, SMALLINT, INT, and BIGINT, can have this property. The Database Engine generates values for such columns sequentially, starting with an initial value. Therefore, you can use the IDENTITY property to let the system generate unique numeric values for the table column of your choice.

Each table can have only one column with the IDENTITY property. The table owner can specify the starting number and the increment value, as shown in Example 6.38.

EXAMPLE 6.38
USE sample;
CREATE TABLE product
  (product_no INTEGER IDENTITY(10000,1) NOT NULL,
   product_name CHAR(30) NOT NULL,
   price MONEY);
SELECT $identity
FROM product
WHERE product_name = 'Soap';

The result could be

<table>
<thead>
<tr>
<th>product_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>10005</td>
</tr>
</tbody>
</table>
The `product` table is created first in Example 6.38. This table has the `product_no` column with the `IDENTITY` property. The values of the `product_no` column are automatically generated by the system, beginning with 10000 and incrementing by 1 for every subsequent value: 10000, 10001, 10002, and so on.

Some system functions and variables are related to the `IDENTITY` property. Example 6.38 uses the `$identity` variable. As the result set of Example 6.38 shows, this variable automatically refers to the column with the `IDENTITY` property.

To find out the starting value and the increment of the column with the `IDENTITY` property, you can use the `IDENT_SEED` and `IDENT_INCR` functions, respectively, in the following way:

```sql
SELECT IDENT_SEED('product'), IDENT_INCR('product')
```

As you already know, the system automatically sets identity values. If you want to supply your own values for particular rows, you must set the `IDENTITY_INSERT` option to ON before the explicit value will be inserted:

```sql
SET IDENTITY_INSERT table_name ON
```

**NOTE**

Because the `IDENTITY_INSERT` option can be used to specify any values for a column with the `IDENTITY` property, `IDENTITY` does not generally enforce uniqueness. Use the `UNIQUE` or `PRIMARY KEY` constraint for this task.

If you insert values after the `IDENTITY_INSERT` option is set to ON, the system presumes that the next value is the incremented value of the highest value that exists in the table at that moment.

**CREATE SEQUENCE Statement**

Using the `IDENTITY` property has several significant disadvantages, the most important of which are the following:

- You can use it only with the specified table.
- You cannot obtain the new value before using it.
- You can specify the `IDENTITY` property only when the column is created.

For these reasons, SQL Server 2012 introduces sequences, which has the same semantics as the `IDENTITY` property but don't have the limitations from the
preceding list. Therefore, a sequence is an independent database feature that enables you to specify a counter of values for different database objects, such as columns and variables. Sequences are created using the CREATE SEQUENCE statement. The CREATE SEQUENCE statement is specified in the SQL standard and is implemented in other relational database systems, such as IBM DB2 and Oracle.

Example 6.39 shows how sequences can be specified in SQL Server.

**EXAMPLE 6.39**

```sql
USE sample;
CREATE SEQUENCE dbo.Sequence1
AS INT
START WITH 1 INCREMENT BY 5
MINVALUE 1 MAXVALUE 256
CYCLE;
```

The values of the sequence called **Sequence1** in Example 6.39 are automatically generated by the system, beginning with 1 and incrementing by 5 for every subsequent value. Therefore, the START clause specifies the initial value, while the INCREMENT clause defines the incremental value. (The incremental value can be positive or negative.)

The following two optional clauses, MINVALUE and MAXVALUE, are directives, which specify a minimal and maximum value for a sequence object. (Note that MINVALUE must be less than or equal to the start value, while MAXVALUE cannot be greater than the upper boundary for the values of the data type used for the specification of the sequence.) The CYCLE clause specifies that the object should restart from the minimum value (or maximum value, for descending sequence objects) when its minimum (or maximum) value is exceeded. The default value for this clause is NO CYCLE, which means that an exception will be thrown if its minimum or maximum value is exceeded.

The main property of a sequence is that it is table-independent; that is, it can be used with any database object, such as a table’s column or variable. (This property positively affects storage and, therefore, performance. You do not need storage for a specified sequence; only the last-used value is stored.)

To generate new sequence values, you can use the NEXT VALUE FOR expression. Example 6.40 shows the use of this expression.

**EXAMPLE 6.40**

```sql
USE sample;
SELECT NEXT VALUE FOR dbo.sequence1;
SELECT NEXT VALUE FOR dbo.sequence1;
```
The result is

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

You can use the NEXT VALUE FOR expression to assign the results of a sequence to a variable or to a column. Example 6.41 shows how you can use this expression to assign the results to a table’s column.

**EXAMPLE 6.41**

```sql
USE sample;
CREATE TABLE dbo.table1
    (column1 INT NOT NULL PRIMARY KEY,
     column2 CHAR(10));
INSERT INTO dbo.table1 VALUES (NEXT VALUE FOR dbo.sequence1, 'A');
INSERT INTO dbo.table1 VALUES (NEXT VALUE FOR dbo.sequence1, 'B');
```

Example 6.41 first creates a table called `table1` with two columns. The following two INSERT statements insert two rows in this table. (For the syntax of the INSERT statement, see Chapter 7.) The first column has values 11 and 16, respectively. (These two values are subsequent values, following the generated values in Example 6.40.)

Example 6.42 shows how you can use the catalog view called `sys.sequences` to check the current value of the sequence, without using it. (Catalog views are described in detail in Chapter 9.)

**EXAMPLE 6.42**

```sql
USE sample;
SELECT current_value
    FROM sys.sequences
    WHERE name = 'Sequence1';
```

**NOTE**

Generally, you use the NEXT VALUE FOR expression in the INSERT statement (see the following chapter) to let the system insert generated values. You can also use the NEXT VALUE FOR expression as part of a multirow query by using the OVER clause (see Example 23.8 in Chapter 23).

The `ALTER SEQUENCE` statement modifies the properties of an existing sequence. One of the most important uses of this statement is in relation to the `RESTART WITH` clause, which “reseeds” a given sequence. Example 6.43 shows the use of the `ALTER SEQUENCE` statement to reinitialize (almost) all properties of the existing sequence called `Sequence1`. 
**EXAMPLE 6.43**

USE sample;
ALTER SEQUENCE dbo.sequence1
    RESTART WITH 100
    INCREMENT BY 50
    MINVALUE 50
    MAXVALUE 200
    NO CYCLE;

To drop a sequence, use the DROP SEQUENCE statement.

**Set Operators**

In addition to the operators described earlier in the chapter, three set operators are supported in the Transact-SQL language:

- UNION
- INTERSECT
- EXCEPT

**UNION Set Operator**

The result of the union of two sets is the set of all elements appearing in either or both of the sets. Accordingly, the union of two tables is a new table consisting of all rows appearing in either or both of the tables.

The general form of the UNION operator is

```
select_1 UNION [ALL] select_2 {
[UNION [ALL] select_3]...}
```

`select_1, select_2,...` are SELECT statements that build the union. If the ALL option is used, all resulting rows, including duplicates, are displayed. The ALL option has the same meaning with the UNION operator as it has in the SELECT list, with one difference: the ALL option is the default in the SELECT list, but it must be specified with the UNION operator to display all resulting rows, including duplicates.

The `sample` database in its original form is not suitable for a demonstration of the UNION operator. For this reason, this section introduces a new table, `employee_enh`, which is identical to the existing `employee` table, up to the additional `domicile` column. The `domicile` column contains the place of residence of every employee.
The new **employee_enh** table has the following form:

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
<th>dept_no</th>
<th>domicile</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>d3</td>
<td>San Antonio</td>
</tr>
<tr>
<td>10102</td>
<td>Ann</td>
<td>Jones</td>
<td>d3</td>
<td>Houston</td>
</tr>
<tr>
<td>18316</td>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
<td>San Antonio</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
<td>d2</td>
<td>Seattle</td>
</tr>
<tr>
<td>9031</td>
<td>Elke</td>
<td>Hansel</td>
<td>d2</td>
<td>Portland</td>
</tr>
<tr>
<td>2581</td>
<td>Elsa</td>
<td>Bertoni</td>
<td>d2</td>
<td>Tacoma</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>d1</td>
<td>Houston</td>
</tr>
</tbody>
</table>

Creation of the **employee_enh** table provides an opportunity to show the use of the INTO clause of the SELECT statement. SELECT INTO has two different parts. First, it creates the new table with the columns corresponding to the columns listed in the SELECT list. Second, it inserts the existing rows of the original table into the new table. (The name of the new table appears with the INTO clause, and the name of the source table appears in the FROM clause of the SELECT statement.)

Example 6.44 shows the creation of the **employee_enh** table.

**EXAMPLE 6.44**

```sql
USE sample;
SELECT  emp_no, emp_fname, emp_lname, dept_no
        INTO employee_enh
    FROM employee;
ALTER TABLE employee_enh
    ADD domicile CHAR(25) NULL;
```

In Example 6.44, SELECT INTO generates the **employee_enh** table and inserts all rows from the initial table (**employee**) into the new one. Finally, the ALTER TABLE statement appends the **domicile** column to the **employee_enh** table.

After the execution of Example 6.44, the **domicile** column contains no values. The values can be added using SQL Server Management Studio (see Chapter 3) or the following UPDATE statements:

```sql
USE sample;
UPDATE employee_enh SET domicile = 'San Antonio'
    WHERE emp_no = 25348;
UPDATE employee_enh SET domicile = 'Houston'
    WHERE emp_no = 10102;
```
Example 6.45 shows the union of the tables `employee_enh` and `department`.

**EXAMPLE 6.45**

```
USE sample;
SELECT domicile
  FROM employee_enh
UNION
SELECT location
  FROM department;
```

The result is

<table>
<thead>
<tr>
<th>domicile</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Antonio</td>
</tr>
<tr>
<td>Houston</td>
</tr>
<tr>
<td>Portland</td>
</tr>
<tr>
<td>Tacoma</td>
</tr>
<tr>
<td>Seattle</td>
</tr>
<tr>
<td>Dallas</td>
</tr>
</tbody>
</table>

Two tables can be connected with the UNION operator if they are compatible with each other. This means that both the SELECT lists must have the same number of columns, and the corresponding columns must have compatible data types. (For example, INT and SMALLINT are compatible data types.)

The ordering of the result of the union can be done only if the ORDER BY clause is used with the last SELECT statement, as shown in Example 6.46. The GROUP BY and HAVING clauses can be used with the particular SELECT statements, but not with the union itself.
EXAMPLE 6.46

Get the employee number for employees who either belong to department d1 or entered their project before 1/1/2007, in ascending order of employee number:

USE sample;
SELECT emp_no
FROM employee
  WHERE dept_no = 'd1'
UNION
SELECT emp_no
FROM works_on
  WHERE enter_date < '01.01.2007'
ORDER BY 1;

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>9031</td>
</tr>
<tr>
<td>10102</td>
</tr>
<tr>
<td>18316</td>
</tr>
<tr>
<td>28559</td>
</tr>
<tr>
<td>29346</td>
</tr>
</tbody>
</table>

NOTE

The UNION operator supports the ALL option. When UNION is used with ALL, duplicates are not removed from the result set.

The OR operator can be used instead of the UNION operator if all SELECT statements connected by one or more UNION operators reference the same table. In this case, the set of the SELECT statements is replaced through one SELECT statement with the set of OR operators.

INTERSECT and EXCEPT Set Operators

The two other set operators are INTERSECT, which specifies the intersection, and EXCEPT, which defines the difference operator. The intersection of two tables is the set of rows belonging to both tables. The difference of two tables is the set of all rows, where the resulting rows belong to the first table but not to the second one. Example 6.47 shows the use of the INTERSECT operator.
**EXAMPLE 6.47**

```
USE sample;
SELECT emp_no
FROM employee
    WHERE dept_no = 'd1'
INTERSECT
SELECT emp_no
FROM works_on
    WHERE enter_date < '01.01.2008';
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>18316</td>
</tr>
<tr>
<td>28559</td>
</tr>
</tbody>
</table>

**NOTE**

*Transact-SQL does not support the INTERSECT operator with the ALL option. (The same is true for the EXCEPT operator.)*

Example 6.48 shows the use of the EXCEPT set operator.

**EXAMPLE 6.48**

```
USE sample;
SELECT emp_no
FROM employee
    WHERE dept_no = 'd3'
EXCEPT
SELECT emp_no
FROM works_on
    WHERE enter_date > '01.01.2008';
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>10102</td>
</tr>
<tr>
<td>25348</td>
</tr>
</tbody>
</table>
**NOTE**

You should be aware that the three set operators have different priorities for evaluation: the INTERSECT operator has the highest priority, EXCEPT is evaluated next, and the UNION operator has the lowest priority. If you do not pay attention to these different priorities, you will get unexpected results when you use several set operators together.

**CASE Expressions**

In database application programming, it is sometimes necessary to modify the representation of data. For instance, a person's gender can be coded using the values 1, 2, and 3 (for female, male, and child, respectively). Such a programming technique can reduce the time for the implementation of a program. The CASE expression in the Transact-SQL language makes this type of encoding easy to implement.

**NOTE**

CASE does not represent a statement (as in most programming languages) but an expression. Therefore, the CASE expression can be used (almost) everywhere where the Transact-SQL language allows the use of an expression.

The CASE expression has two different forms:

- Simple CASE expression
- Searched CASE expression

The syntax of the simple CASE expression is

```sql
CASE expression_1
  {WHEN expression_2 THEN result_1} ...
  [ELSE result_n]
END
```

A Transact-SQL statement with the simple CASE expression looks for the first expression in the list of all WHEN clauses that match `expression_1` and evaluates the corresponding THEN clause. If there is no match, the ELSE clause is evaluated.

The syntax of the searched CASE expression is

```sql
CASE
  {WHEN condition_1 THEN result_1} ...
  [ELSE result_n]
END
```
A Transact-SQL statement with the searched CASE expression looks for the first expression that evaluates to TRUE. If none of the WHEN conditions evaluates to TRUE, the value of the ELSE expression is returned. Example 6.49 shows the use of the searched CASE expression.

**EXAMPLE 6.49**

```sql
USE sample;
SELECT project_name,
    CASE
        WHEN budget > 0 AND budget < 100000  THEN 1
        WHEN budget >= 100000 AND budget < 200000  THEN 2
        WHEN budget >= 200000 AND budget < 300000  THEN 3
        ELSE 4
    END budget_weight
FROM project;
```

The result is

<table>
<thead>
<tr>
<th>project_name</th>
<th>budget_weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apollo</td>
<td>2</td>
</tr>
<tr>
<td>Gemini</td>
<td>1</td>
</tr>
<tr>
<td>Mercury</td>
<td>2</td>
</tr>
</tbody>
</table>

In Example 6.49, budgets of all projects are weighted, and the calculated weights (together with the name of the corresponding project) are displayed.

Example 6.50 shows another example with the CASE expression, where the WHEN clause contains inner queries as part of the expression.

**EXAMPLE 6.50**

```sql
USE sample;
SELECT project_name,
    CASE
        WHEN p1.budget < (SELECT AVG(p2.budget) FROM project p2) THEN 'below average'
        WHEN p1.budget = (SELECT AVG(p2.budget) FROM project p2) THEN 'on average'
        WHEN p1.budget > (SELECT AVG(p2.budget) FROM project p2) THEN 'above average'
    END budget_category
FROM project p1;
```
The result is

<table>
<thead>
<tr>
<th>project_name</th>
<th>budget_category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apollo</td>
<td>below average</td>
</tr>
<tr>
<td>Gemini</td>
<td>below average</td>
</tr>
<tr>
<td>Mercury</td>
<td>above average</td>
</tr>
</tbody>
</table>

**Subqueries**

All previous examples in this chapter contain comparisons of column values with an expression, constant, or set of constants. Additionally, the Transact-SQL language offers the ability to compare column values with the result of another SELECT statement. Such a construct, where one or more SELECT statements are nested in the WHERE clause of another SELECT statement, is called a subquery. The first SELECT statement of a subquery is called the outer query—in contrast to the inner query, which denotes the SELECT statement(s) used in a comparison. The inner query will be evaluated first, and the outer query receives the values of the inner query.

**NOTE**

An inner query can also be nested in an INSERT, UPDATE, or DELETE statement, which will be discussed later in this book.

There are two types of subqueries:

- Self-contained
- Correlated

In a self-contained subquery, the inner query is logically evaluated exactly once. A correlated subquery differs from a self-contained one in that its value depends upon a variable from the outer query. Therefore, the inner query of a correlated subquery is logically evaluated each time the system retrieves a new row from the outer query. This section shows examples of self-contained subqueries. The correlated subquery will be discussed later in the chapter, together with the join operation.
A self-contained subquery can be used with the following operators:

- Comparison operators
- IN operator
- ANY or ALL operator

### Subqueries and Comparison Operators

Example 6.51 shows the self-contained subquery that is used with the operator =.

**EXAMPLE 6.51**

Get the first and last names of employees who work in the Research department:

```sql
USE sample;
SELECT emp_fname, emp_lname
FROM employee
WHERE dept_no = (SELECT dept_no
                   FROM department
                   WHERE dept_name = 'Research');
```

The result is

<table>
<thead>
<tr>
<th>emp_fname</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Barrimore</td>
</tr>
<tr>
<td>Sybill</td>
<td>Moser</td>
</tr>
</tbody>
</table>

The inner query of Example 6.51 is logically evaluated first. That query returns the number of the research department (d1). Thus, after the evaluation of the inner query, the subquery in Example 6.51 can be represented with the following equivalent query:

```sql
USE sample
SELECT emp_fname, emp_lname
FROM employee
WHERE dept_no = 'd1';
```

A subquery can be used with other comparison operators, too. Any comparison operator can be used, provided that the inner query returns exactly one row. This is
obvious because comparison between particular column values of the outer query and a set of values (as a result of the inner query) is not possible. The following section shows how you can handle the case in which the result of an inner query contains a set of values.

Subqueries and the IN Operator

The IN operator allows the specification of a set of expressions (or constants) that are subsequently used for the query search. This operator can be applied to a subquery for the same reason—that is, when the result of an inner query contains a set of values.

Example 6.52 shows the use of the IN operator in a subquery.

EXAMPLE 6.52

Get full details of all employees whose department is located in Dallas:

USE sample;
SELECT * FROM employee
WHERE dept_no IN (SELECT dept_no FROM department
WHERE location = 'Dallas');

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
<th>dept_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>d3</td>
</tr>
<tr>
<td>10102</td>
<td>Ann</td>
<td>Jones</td>
<td>d3</td>
</tr>
<tr>
<td>18316</td>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>d1</td>
</tr>
</tbody>
</table>

Each inner query may contain further queries. This type of subquery is called a subquery with multiple levels of nesting. The maximum number of inner queries in a subquery depends on the amount of memory the Database Engine has for each SELECT statement. In the case of subqueries with multiple levels of nesting, the system first evaluates the innermost query and returns the result to the query on the next nesting level, and so on. Finally, the outermost query evaluates the final outcome.

Example 6.53 shows the query with multiple levels of nesting.
**EXAMPLE 6.53**

Get the last names of all employees who work on the project Apollo:

```
USE sample;
SELECT emp_lname
FROM employee
WHERE emp_no IN
(SELECT emp_no
  FROM works_on
  WHERE project_no IN
    (SELECT project_no
     FROM project
     WHERE project_name = 'Apollo'));
```

The result is

<table>
<thead>
<tr>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>James</td>
</tr>
<tr>
<td>Bertoni</td>
</tr>
<tr>
<td>Moser</td>
</tr>
</tbody>
</table>

The innermost query in Example 6.53 evaluates to the `project_no` value p1. The middle inner query compares this value with all values of the `project_no` column in the `works_on` table. The result of this query is the set of employee numbers: (10102, 29346, 9031, 28559). Finally, the outermost query displays the corresponding last names for the selected employee numbers.

**Subqueries and ANY and ALL Operators**

The operators ANY and ALL are always used in combination with one of the comparison operators. The general syntax of both operators is

```
column_name operator [ANY | ALL] query
```

where `operator` stands for a comparison operator and `query` is an inner query.

The ANY operator evaluates to TRUE if the result of the corresponding inner query contains at least one row that satisfies the comparison. The keyword SOME is the synonym for ANY. Example 6.54 shows the use of the ANY operator.
EXAMPLE 6.54

Get the employee numbers, project numbers, and job names for employees who have not spent the most time on one of the projects:

USE sample;
SELECT DISTINCT emp_no, project_no, job
FROM works_on
WHERE enter_date > ANY
(SELECT enter_date
FROM works_on);

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>project_no</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>2581</td>
<td>p3</td>
<td>Analyst</td>
</tr>
<tr>
<td>9031</td>
<td>p1</td>
<td>Manager</td>
</tr>
<tr>
<td>9031</td>
<td>p3</td>
<td>Clerk</td>
</tr>
<tr>
<td>10102</td>
<td>p3</td>
<td>Manager</td>
</tr>
<tr>
<td>18316</td>
<td>p2</td>
<td>NULL</td>
</tr>
<tr>
<td>25348</td>
<td>p2</td>
<td>Clerk</td>
</tr>
<tr>
<td>28559</td>
<td>p1</td>
<td>NULL</td>
</tr>
<tr>
<td>28559</td>
<td>p2</td>
<td>Clerk</td>
</tr>
<tr>
<td>29346</td>
<td>p1</td>
<td>Clerk</td>
</tr>
<tr>
<td>29346</td>
<td>p2</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Each value of the enter_date column in Example 6.54 is compared with all values of this column. For all dates of the column, except the oldest one, the comparison is evaluated to TRUE at least once. The row with the oldest date does not belong to the result because the comparison does not evaluate to TRUE in any case. In other words, the expression “enter_date > ANY (SELECT enter_date FROM works_on)” is true if there are any (one or more) rows in the works_on table with a value of the enter_date column less than the value of enter_date for the current row. This will be true for all but the earliest value of the enter_date column.

The ALL operator evaluates to TRUE if the evaluation of the table column in the inner query returns all values of that column.
Note
Do not use ANY and ALL operators! Every query using ANY or ALL can be better formulated with the EXISTS function, which is explained later in this chapter (see the section “Subqueries and the EXISTS Function”). Additionally, the semantic meaning of the ANY operator can be easily confused with the semantic meaning of the ALL operator, and vice versa.

Temporary Tables
A temporary table is a database object that is temporarily stored and managed by the database system. Temporary tables can be local or global. Local temporary tables have physical representation—that is, they are stored in the tempdb system database. They are specified with the prefix # (for example, #table_name).

A local temporary table is owned by the session that created it and is visible only to that session. Such a table is thus automatically dropped when the creating session terminates. (If you define a local temporary table inside a stored procedure, it will be destroyed when the corresponding procedure terminates.)

Global temporary tables are visible to any user and any connection after they are created, and are deleted when all users that are referencing the table disconnect from the database server. In contrast to local temporary tables, global ones are specified with the prefix ##.

Examples 6.55 and 6.56 show how the temporary table project_temp can be created using two different Transact-SQL statements.

Example 6.55
USE sample;
CREATE TABLE #project_temp
    (project_no CHAR(4) NOT NULL,
     project_name CHAR(25) NOT NULL);

Example 6.56
USE sample;
SELECT project_no, project_name
    INTO #project_temp1
    FROM project;

Examples 6.55 and 6.56 are similar. They use two different Transact-SQL statements to create the local temporary table, #project_temp and #project_temp1, respectively. However, Example 6.55 leaves it empty, while Example 6.56 populates the temporary table with the data from the project table.
Join Operator

The previous sections of this chapter demonstrated the use of the SELECT statement to query rows from one table of a database. If the Transact-SQL language supported only such simple SELECT statements, the attachment of two or more tables to retrieve data would not be possible. Consequently, all data of a database would have to be stored in one table. Although the storage of all the data of a database inside one table is possible, it has one main disadvantage—the stored data are highly redundant.

Transact-SQL provides the join operator, which retrieves data from more than one table. This operator is probably the most important operator for relational database systems, because it allows data to be spread over many tables and thus achieves a vital property of database systems—nonredundant data.

**NOTE**

The UNION operator also attaches two or more tables. However, the UNION operator always attaches two or more SELECT statements, while the join operator “joins” two or more tables using just one SELECT. Further, the UNION operator attaches rows of tables, while, as you will see later in this section, the join operator “joins” columns of tables.

The join operator is applied to base tables and views. In this chapter, joins between base tables are discussed, while joins concerning views will be discussed in Chapter 11.

There are several different forms of the join operator. This section discusses the following fundamental types:

- Natural join
- Cartesian product (cross join)
- Outer join
- Theta join, self-join, and semi-join

Before explaining different join forms, this section describes the different syntax forms of the join operator.

**Two Syntax Forms to Implement Joins**

To join tables, you can use two different forms:

- Explicit join syntax (ANSI SQL:1992 join syntax)
- Implicit join syntax (old-style join syntax)
Chapter 6: Queries

The ANSI SQL:1992 join syntax was introduced in the SQL92 standard and defines join operations explicitly—that is, using the corresponding name for each type of join operation. The keywords concerning the explicit definition of join are

- CROSS JOIN
- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN

CROSS JOIN specifies the Cartesian product of two tables. INNER JOIN defines the natural join of two tables, while LEFT OUTER JOIN and RIGHT OUTER JOIN characterize the join operations of the same names, respectively. Finally, FULL OUTER JOIN specifies the union of the right and left outer joins. (All these different join operations are explained in the following sections.)

The implicit join syntax is “old-style” syntax, where each join operation is defined implicitly via the WHERE clause, using the so-called join columns (see the second statement in Example 6.57).

**NOTE**

Use of the explicit join syntax is recommended. This syntax enhances the readability of queries. For this reason, all examples in this chapter concerning the join operation are solved using the explicit syntax forms. In a few introductory examples, you will see the old-style syntax, too.

**Natural Join**

*Natural join* is best explained through the use of an example, so check out Example 6.57.

**NOTE**

The phrases “natural join” and “equi-join” are often used as synonyms, but there is a slight difference between them. The equi-join operation always has one or more pairs of columns that have identical values in every row. The operation that eliminates such columns from the equi-join is called a natural join.

**EXAMPLE 6.57**

Get full details of each employee; that is, besides the employee’s number, first and last names, and corresponding department number, also get the name of his or her department and its location, with duplicate columns displayed.
The following is the explicit join syntax:

```
USE sample;
SELECT employee.*, department.*
FROM employee INNER JOIN department
ON employee.dept_no = department.dept_no;
```

The SELECT list in Example 6.57 includes all columns of the **employee** and **department** tables. The FROM clause in the SELECT statement specifies the tables that are joined as well as the explicit name of the join form (INNER JOIN). The ON clause is also part of the FROM clause; it specifies the join columns from both tables. The condition `employee.dept_no = department.dept_no` specifies a **join condition**, and both columns are said to be **join columns**.

The equivalent solution with the old-style, implicit join syntax is as follows:

```
USE sample;
SELECT employee.*, department.*
FROM employee, department
WHERE employee.dept_no = department.dept_no;
```

This syntax has two significant differences from the explicit join syntax: the FROM clause of the query contains the list of tables that are joined, and the corresponding join condition is specified in the WHERE clause using join columns.

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_f_name</th>
<th>emp_l_name</th>
<th>dept_no</th>
<th>dept_no</th>
<th>dept_name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>d3</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>10102</td>
<td>Ann</td>
<td>Jones</td>
<td>d3</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>18316</td>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
<td>d2</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>9031</td>
<td>Elsa</td>
<td>Bertoni</td>
<td>d2</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>2581</td>
<td>Elke</td>
<td>Hansel</td>
<td>d2</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>d1</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
</tbody>
</table>

**NOTE**

It is strongly recommended that you use * in a SELECT list only when you are using interactive SQL, and avoid its use in an application program.
Example 6.57 can be used to show how a join operation works. Note that this is just an illustration of how you can think about the join process; the Database Engine actually has several strategies from which it chooses to implement the join operator. Imagine each row of the employee table combined with each row of the department table. The result of this combination is a table with 7 columns (4 from the table employee and 3 from the table department) and 21 rows (see Table 6-1).

In the second step, all rows from Table 6-1 that do not satisfy the join condition employee.dept_no = department.dept_no are removed. These rows are prefixed in Table 6-1 with the * sign. The rest of the rows represent the result of Example 6.57.

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
<th>dept_no</th>
<th>dept_no</th>
<th>dept_name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>*25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>d3</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>*10102</td>
<td>Ann</td>
<td>Jones</td>
<td>d3</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>18316</td>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>*29346</td>
<td>James</td>
<td>James</td>
<td>d2</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>*9031</td>
<td>Elsa</td>
<td>Bertoni</td>
<td>d2</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>*2581</td>
<td>Elke</td>
<td>Hansel</td>
<td>d2</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>d1</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>*25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>d3</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>*10102</td>
<td>Ann</td>
<td>Jones</td>
<td>d3</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>*18316</td>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
<td>d2</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>9031</td>
<td>Elsa</td>
<td>Bertoni</td>
<td>d2</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>2581</td>
<td>Elke</td>
<td>Hansel</td>
<td>d2</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>*28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>d1</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>d3</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>10102</td>
<td>Ann</td>
<td>Jones</td>
<td>d3</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>*18316</td>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>*29346</td>
<td>James</td>
<td>James</td>
<td>d2</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>*9031</td>
<td>Elsa</td>
<td>Bertoni</td>
<td>d2</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>*2581</td>
<td>Elke</td>
<td>Hansel</td>
<td>d2</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>*28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>d1</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
</tbody>
</table>

Table 6-1 Result of the Cartesian Product Between the Tables employee and department
The semantics of the corresponding join columns must be identical. This means both columns must have the same logical meaning. It is not required that the corresponding join columns have the same name (or even an identical type), although this will often be the case.

**NOTE**

It is not possible for a database system to check the logical meaning of a column. (For instance, project number and employee number have nothing in common, although both columns are defined as integers.) Therefore, database systems can check only the data type and the length of string data types. The Database Engine requires that the corresponding join columns have compatible data types, such as INT and SMALLINT.

The sample database contains three pairs of columns in which each column of the pair has the same logical meaning (and they have the same names as well). The employee and department tables can be joined using the columns employee.dept_no and department.dept_no. The join columns of the employee and works_on tables are the columns employee.emp_no and works_on.emp_no. Finally, the project and works_on tables can be joined using the join columns project.project_no and works_on.project_no.

The names of columns in a SELECT statement can be qualified. “Qualifying” a column name means that, to avoid any possible ambiguity about which table the column belongs to, the column name is preceded by its table name (or the alias of the table), separated by a period: `table_name.column_name`.

In most SELECT statements a column name does not need any qualification, although the use of qualified names is generally recommended for readability. If column names within a SELECT statement are ambiguous (like the columns employee.dept_no and department.dept_no in Example 6.57), the qualified names for the columns must be used.

In a SELECT statement with a join, the WHERE clause can include other conditions in addition to the join condition, as shown in Example 6.58.

**EXAMPLE 6.58**

Get full details of all employees who work on the project Gemini.

Explicit join syntax:

```sql
USE sample;
SELECT  emp_no, project.project_no, job, enter_date, project_name, budget
FROM works_on JOIN project
ON project.project_no = works_on.project_no
WHERE project_name = 'Gemini';
```
Old-style join syntax:

USE sample;
SELECT emp_no, project.project_no, job, enter_date, project_name, budget
FROM works_on, project
WHERE project.project_no = works_on.project_no
AND project_name = 'Gemini';

NOTE
The qualification of the columns emp_no, project_name, job, and budget in Example 6.58 is not necessary, because there is no ambiguity regarding these names.

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>project_no</th>
<th>job</th>
<th>enter_date</th>
<th>project_name</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>p2</td>
<td>Clerk</td>
<td>2007-02-15</td>
<td>Gemini</td>
<td>95000.0</td>
</tr>
<tr>
<td>18316</td>
<td>p2</td>
<td>NULL</td>
<td>2007-06-01</td>
<td>Gemini</td>
<td>95000.0</td>
</tr>
<tr>
<td>29346</td>
<td>p2</td>
<td>NULL</td>
<td>2006-12-15</td>
<td>Gemini</td>
<td>95000.0</td>
</tr>
<tr>
<td>28559</td>
<td>p2</td>
<td>Clerk</td>
<td>2008-02-01</td>
<td>Gemini</td>
<td>95000.0</td>
</tr>
</tbody>
</table>

From this point forward, all examples will be implemented using the explicit join syntax only.

Example 6.59 shows another use of the inner join.

EXAMPLE 6.59

Get the department number for all employees who entered their projects on October 15, 2007:

USE sample;
SELECT dept_no
FROM employee JOIN works_on
ON employee.emp_no = works_on.emp_no
WHERE enter_date = '10.15.2007';

The result is

dept_no
d2
Joining More Than Two Tables

Theoretically, there is no upper limit on the number of tables that can be joined using a SELECT statement. (One join condition always combines two tables!) However, the Database Engine has an implementation restriction: the maximum number of tables that can be joined in a SELECT statement is 64.

Example 6.60 joins three tables of the sample database.

**EXAMPLE 6.60**

Get the first and last names of all analysts whose department is located in Seattle:

```sql
USE sample;
SELECT emp_fname, emp_lname
FROM works_on JOIN employee
    ON works_on.emp_no = employee.emp_no
    JOIN department
    ON employee.dept_no = department.dept_no
    AND location = 'Seattle'
    AND job = 'analyst';
```

The result is

<table>
<thead>
<tr>
<th>emp_fname</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elke</td>
<td>Hansel</td>
</tr>
</tbody>
</table>

The result in Example 6.60 can be obtained only if you join at least three tables: `works_on`, `employee`, and `department`. These tables can be joined using two pairs of join columns:

- `(works_on.emp_no, employee.emp_no)`
- `(employee.dept_no, department.dept_no)`

Example 6.61 uses all four tables from the sample database to obtain the result set.

**EXAMPLE 6.61**

Get the names of projects (with redundant duplicates eliminated) being worked on by employees in the Accounting department:

```sql
USE sample;
SELECT DISTINCT project_name
FROM project JOIN works_on
    ON project.project_no = works_on.project_no
    JOIN employee
    ON works_on.emp_no = employee.emp_no
```
JOIN department ON employee.dept_no = department.dept_no
WHERE dept_name = 'Accounting';

The result is

<table>
<thead>
<tr>
<th>project_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apollo</td>
</tr>
<tr>
<td>Gemini</td>
</tr>
<tr>
<td>Mercury</td>
</tr>
</tbody>
</table>

Notice that when joining three tables, you use two join conditions (linking two
tables each) to achieve a natural join. When you join four tables, you use three such
join conditions. In general, if you join \( n \) tables, you need \( n - 1 \) join conditions to avoid
a Cartesian product. Of course, using more than \( n - 1 \) join conditions, as well as other
conditions, is certainly permissible to further reduce the result set.

**Cartesian Product**

The previous section illustrated a possible method of producing a natural join. In the
first step of this process, each row of the `employee` table is combined with each row
of the `department` table. This intermediate result was made by the operation called
*Cartesian product*. Example 6.62 shows the Cartesian product of the tables `employee`
and `department`.

**EXAMPLE 6.62**

```sql
USE sample;
SELECT employee.*, department.*
FROM employee CROSS JOIN department;
```

The result of Example 6.62 is shown in Table 6-1. A Cartesian product combines
each row of the first table with each row of the second table. In general, the Cartesian
product of two tables such that the first table has \( n \) rows and the second table has
\( m \) rows will produce a result with \( n \) times \( m \) rows (or \( n \cdot m \)). Thus, the result set in
Example 6.62 contains \( 7 \cdot 3 = 21 \) rows.

In practice, the use of a Cartesian product is highly unusual. Sometimes users
generate the Cartesian product of two tables when they forget to include the join
condition in the WHERE clause of the old-style join syntax. In this case, the output
does not correspond to the expected result because it contains too many rows. (The
existence of many and unexpected rows in the result is a hint that a Cartesian product
of two tables, rather than the intended natural join, has been produced.)
Outer Join

In the previous examples of natural join, the result set included only rows from one table that have corresponding rows in the other table. Sometimes it is necessary to retrieve, in addition to the matching rows, the unmatched rows from one or both of the tables. Such an operation is called an outer join.

Examples 6.63 and 6.64 show the difference between a natural join and the corresponding outer join. (All examples in this section use the `employee_enh` table.)

**EXAMPLE 6.63**

Get full details of all employees, including the location of their department, who live and work in the same city:

```sql
USE sample;
SELECT employee_enh.*, department.location
FROM employee_enh JOIN department
ON domicile = location;
```

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
<th>dept_no</th>
<th>domicile</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
<td>d2</td>
<td>Seattle</td>
<td>Seattle</td>
</tr>
</tbody>
</table>

Example 6.63 uses a natural join to display the result set of rows. If you would like to know all other existing living places of employees, you have to use the (left) outer join. This is called a left outer join because all rows from the table on the left side of the operator are returned, whether or not they have a matching row in the table on the right. In other words, if there are no matching rows in the table on the right side, the outer join will still return a row from the table on the left side, with NULL in each column of the other table (see Example 6.64). The Database Engine uses the operator LEFT OUTER JOIN to specify the left outer join.

A right outer join is similar, but it returns all rows of the table on the right of the symbol. The Database Engine uses the operator RIGHT OUTER JOIN to specify the right outer join.

**EXAMPLE 6.64**

Get full details of all employees, including the location of their department, for all cities that are either the living place only or both the living and working place of employees:

```sql
USE sample;
SELECT employee_enh.*, department.location
FROM employee_enh LEFT OUTER JOIN department
ON domicile = location;
```
The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp lname</th>
<th>dept_no</th>
<th>domicile</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>d3</td>
<td>San Antonio</td>
<td>NULL</td>
</tr>
<tr>
<td>10102</td>
<td>Ann</td>
<td>Jones</td>
<td>d3</td>
<td>Houston</td>
<td>NULL</td>
</tr>
<tr>
<td>18316</td>
<td>John</td>
<td>Barrimore</td>
<td>d1</td>
<td>San Antonio</td>
<td>NULL</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
<td>d2</td>
<td>Seattle</td>
<td>Seattle</td>
</tr>
<tr>
<td>9031</td>
<td>Elsa</td>
<td>Bertoni</td>
<td>d2</td>
<td>Portland</td>
<td>NULL</td>
</tr>
<tr>
<td>2581</td>
<td>Elke</td>
<td>Hansel</td>
<td>d2</td>
<td>Tacoma</td>
<td>NULL</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>d1</td>
<td>Houston</td>
<td>NULL</td>
</tr>
</tbody>
</table>

As you can see, when there is no matching row in the table on the right side (department, in this case), the left outer join still returns the rows from the table on the left side (employee_enh), and the columns of the other table are populated by NULL values. Example 6.65 shows the use of the right outer join operation.

**EXAMPLE 6.65**

Get full details of all departments, as well as all living places of their employees, for all cities that are either the location of a department or the living and working place of an employee:

```sql
USE sample;
SELECT employee_enh.domicile, department.*
FROM employee_enh RIGHT OUTER JOIN department
ON domicile = location;
```

The result is

<table>
<thead>
<tr>
<th>domicile</th>
<th>dept_no</th>
<th>dept_name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td>d2</td>
<td>Accounting</td>
<td>Seattle</td>
</tr>
<tr>
<td>NULL</td>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>NULL</td>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
</tbody>
</table>

In addition to the left and right outer joins, there is also the full outer join, which is defined as the union of the left and right outer joins. In other words, all rows from both tables are represented in the result set. If there is no corresponding row in one of the tables, its columns are returned with NULL values. This operation is specified using the FULL OUTER JOIN operator.
Every outer join operation can be simulated using the UNION operator plus the NOT EXISTS function. Example 6.66 is equivalent to the example with the left outer join (Example 6.64).

**EXAMPLE 6.66**
Get full details of all employees, including the location of their department, for all cities that are either the living place only or both the living and working place of employees:

```
USE sample;
SELECT employee_enh.*, department.location
FROM employee_enh JOIN department
ON domicile = location
UNION
SELECT employee_enh.*, 'NULL'
FROM employee_enh
WHERE NOT EXISTS
(SELECT *
    FROM department
    WHERE location = domicile);
```

The first SELECT statement in the union specifies the natural join of the tables **employee_enh** and **department** with the join columns **domicile** and **location**. This SELECT statement retrieves all cities that are at the same time the living places and working places of each employee. The second SELECT statement in the union retrieves, additionally, all rows from the **employee_enh** table that do not match the condition in the natural join.

**Further Forms of Join Operations**
The preceding sections discussed the most important join forms. This section shows you three other forms:

- Theta join
- Self-join
- Semi-join

The following subsections describe these forms.

**Theta Join**
Join columns need not be compared using the equality sign. A join operation using a general join condition is called a theta join. Example 6.67, which uses the **employee_enh** table, shows the theta join operation.
**EXAMPLE 6.67**

Get all the combinations of employee information and department information where the domicile of an employee alphabetically precedes any location of departments.

```
USE sample;
SELECT emp_fname, emp_lname, domicile, location
FROM employee_enh JOIN department
    ON domicile < location;
```

The result is

<table>
<thead>
<tr>
<th>emp_fname</th>
<th>emp_lname</th>
<th>domicile</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matthew</td>
<td>Smith</td>
<td>San Antonio</td>
<td>Seattle</td>
</tr>
<tr>
<td>Ann</td>
<td>Jones</td>
<td>Houston</td>
<td>Seattle</td>
</tr>
<tr>
<td>John</td>
<td>Barrimore</td>
<td>San Antonio</td>
<td>Seattle</td>
</tr>
<tr>
<td>Elsa</td>
<td>Bertoni</td>
<td>Portland</td>
<td>Seattle</td>
</tr>
<tr>
<td>Sybill</td>
<td>Moser</td>
<td>Houston</td>
<td>Seattle</td>
</tr>
</tbody>
</table>

In Example 6.67, the corresponding values of columns `domicile` and `location` are compared. In every resulting row, the value of the `domicile` column is ordered alphabetically before the corresponding value of the `location` column.

**Self-Join, or Joining a Table with Itself**

In addition to joining two or more different tables, a natural join operation can also be applied to a single table. In this case, the table is joined with itself, whereby a single column of the table is compared with itself. The comparison of a column with itself means that the table name appears twice in the FROM clause of a SELECT statement. Therefore, you need to be able to reference the name of the same table twice. This can be accomplished using at least one alias name. The same is true for the column names in the join condition of a SELECT statement. In order to distinguish both column names, you use the qualified names. Example 6.68 joins the `department` table with itself.

**EXAMPLE 6.68**

Get full details of all departments located at the same location as at least one other department:

```
USE sample;
SELECT t1.dept_no, t1.dept_name, t1.location
FROM department t1 JOIN department t2
```
ON  t1.location = t2.location
WHERE t1.dept_no <> t2.dept_no;

The result is

<table>
<thead>
<tr>
<th>dept_no</th>
<th>dept_name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
<tr>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
</tbody>
</table>

The FROM clause in Example 6.68 contains two aliases for the department table: t1 and t2. The first condition in the WHERE clause specifies the join columns, while the second condition eliminates unnecessary duplicates by making certain that each department is compared with different departments.

**Semi-Join**

The semi-join is similar to the natural join, but the result of the semi-join is only the set of all rows from one table where one or more matches are found in the second table. Example 6.69 shows the semi-join operation.

**EXAMPLE 6.69**

USE sample;
SELECT emp_no, emp_lname, e.dept_no
  FROM employee e JOIN department d
  ON e.dept_no = d.dept_no
  WHERE location = 'Dallas';

The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_lname</th>
<th>dept_no</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Smith</td>
<td>d3</td>
</tr>
<tr>
<td>10102</td>
<td>Jones</td>
<td>d3</td>
</tr>
<tr>
<td>18316</td>
<td>Barrimore</td>
<td>d1</td>
</tr>
<tr>
<td>28559</td>
<td>Moser</td>
<td>d1</td>
</tr>
</tbody>
</table>

As can be seen from Example 6.69, the SELECT list of the semi-join contains only columns from the employee table. This is exactly what characterizes the semi-join operation. This operation is usually used in distributed query processing to minimize data transfer. The Database Engine uses the semi-join operation to implement the feature called star join (see Chapter 25).
Correlated Subqueries

A subquery is said to be a *correlated subquery* if the inner query depends on the outer query for any of its values. Example 6.70 shows a correlated subquery.

**Example 6.70**

Get the last names of all employees who work on project p3:

```
USE sample;
SELECT emp_lname
FROM employee
WHERE 'p3' IN
(SELECT project_no
 FROM works_on
 WHERE works_on.emp_no = employee.emp_no);
```

The result is

<table>
<thead>
<tr>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Bertoni</td>
</tr>
<tr>
<td>Hansel</td>
</tr>
</tbody>
</table>

The inner query in Example 6.70 must be logically evaluated many times because it contains the `emp_no` column, which belongs to the `employee` table in the outer query, and the value of the `emp_no` column changes every time the Database Engine examines a different row of the `employee` table in the outer query.

Let’s walk through how the system might process the query in Example 6.70. First, the system retrieves the first row of the `employee` table (for the outer query) and compares the employee number of that column (25348) with values of the `works_on.emp_no` column in the inner query. Since the only `project_no` for this employee is p2, the inner query returns the value p2. The single value in the set is not equal to the constant value p3 in the outer query, so the outer query’s condition (WHERE ‘p3’ IN …) is not met and no rows are returned by the outer query for this employee. Then, the system retrieves the next row of the `employee` table and repeats the comparison of employee numbers in both tables. The second employee has two rows in the `works_on` table with `project_no` values of p1 and p3, so the result set of the inner query is (p1,p3). One of the elements in the result set is equal to the constant value p3, so the condition is evaluated to TRUE and the corresponding value of the `emp_lname` column in the second row (Jones) is displayed.

The same process is applied to all rows of the `employee` table, and the final result set with three rows is retrieved.

More examples of correlated subqueries are shown in the next section.
**Subqueries and the EXISTS Function**

The EXISTS function takes an inner query as an argument and returns TRUE if the inner query returns one or more rows, and returns FALSE if it returns zero rows. This function will be explained using examples, starting with Example 6.71.

**EXAMPLE 6.71**

Get the last names of all employees who work on project p1:

```
USE sample;
SELECT emp_lname
FROM employee
WHERE EXISTS
(SELECT *
    FROM works_on
    WHERE employee.emp_no = works_on.emp_no
    AND project_no = 'p1');
```

The result is

<table>
<thead>
<tr>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>James</td>
</tr>
<tr>
<td>Bertoni</td>
</tr>
<tr>
<td>Moser</td>
</tr>
</tbody>
</table>

The inner query of the EXISTS function almost always depends on a variable from an outer query. Therefore, the EXISTS function usually specifies a correlated subquery.

Let’s walk through how the Database Engine might process the query in Example 6.71. First, the outer query considers the first row of the **employee** table (Smith). Next, the EXISTS function is evaluated to determine whether there are any rows in the **works_on** table whose employee number matches the one from the current row in the outer query, and whose **project_no** is p1. Because Mr. Smith does not work on the project p1, the result of the inner query is an empty set and the EXISTS function is evaluated to FALSE. Therefore, the employee named Smith does not belong to the final result set. Using this process, all rows of the **employee** table are tested, and the result set is displayed.

Example 6.72 shows the use of the NOT EXISTS function.
EXAMPLE 6.72

Get the last names of all employees who work for departments not located in Seattle:

USE sample;
SELECT emp_lname
FROM employee
WHERE NOT EXISTS
  (SELECT *
   FROM department
   WHERE employee.dept_no = department.dept_no
   AND location = 'Seattle');

The result is

<table>
<thead>
<tr>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
</tr>
<tr>
<td>Jones</td>
</tr>
<tr>
<td>Barrimore</td>
</tr>
<tr>
<td>Moser</td>
</tr>
</tbody>
</table>

The SELECT list of an outer query involving the EXISTS function is not required to be of the form SELECT * as in the previous examples. The form SELECT column_list, where column_list is one or more columns of the table, is an alternate form. Both forms are equivalent, because the EXISTS function tests only the existence (i.e., nonexistence) of rows in the result set. For this reason, the use of SELECT * in this case is safe.

Should You Use Joins or Subqueries?

Almost all SELECT statements that join tables and use the join operator can be rewritten as subqueries, and vice versa. Writing the SELECT statement using the join operator is often easier to read and understand and can also help the Database Engine to find a more efficient strategy for retrieving the appropriate data. However, there are a few problems that can be easier solved using subqueries, and there are others that can be easier solved using joins.

Subquery Advantages

Subqueries are advantageous over joins when you have to calculate an aggregate value on-the-fly and use it in the outer query for comparison. Example 6.73 shows this.
EXAMPLE 6.73

Get the employee numbers and enter dates of all employees with enter dates equal to the earliest date:

USE sample;
SELECT emp_no, enter_date
FROM works_on
WHERE enter_date = (SELECT min(enter_date)
FROM works_on);

This problem cannot be solved easily with a join, because you would have to write the aggregate function in the WHERE clause, which is not allowed. (You can solve the problem using two separate queries in relation to the works_on table.)

Join Advantages

Joins are advantageous over subqueries if the SELECT list in a query contains columns from more than one table. Example 6.74 shows this.

EXAMPLE 6.74

Get the employee numbers, last names, and jobs for all employees who entered their projects on October 15, 2007:

USE sample;
SELECT employee.emp_no, emp_lname, job
FROM employee, works_on
WHERE employee.emp_no = works_on.emp_no
AND enter_date = '10.15.2007';

The SELECT list of the query in Example 6.74 contains columns emp_no and emp_lname from the employee table and the job column from the works_on table. For this reason, the equivalent solution with the subquery would display an error, because subqueries can display information only from the outer table.

Table Expressions

Table expressions are subqueries that are used where a table is expected. There are two types of table expressions:
Derived tables

Derived Tables

A derived table is a table expression that appears in the FROM clause of a query. You can apply derived tables when the use of column aliases is not possible because another clause is processed by the SQL translator before the alias name is known. Example 6.75 shows an attempt to use a column alias where another clause is processed before the alias name is known.

**EXAMPLE 6.75 (EXAMPLE OF AN ILLEGAL STATEMENT)**

Get all existing groups of months from the `enter_date` column of the `works_on` table:

```
USE sample;
SELECT MONTH(enter_date) as enter_month
FROM works_on
GROUP BY enter_month;
```

The result is

```
Message 207: Level 16, State 1, Line 4
The invalid column 'enter_month'
```

The reason for the error message is that the GROUP BY clause is processed before the corresponding SELECT list, and the alias name `enter_month` is not known at the time the grouping is processed.

By using a derived table that contains the preceding query (without the GROUP BY clause), you can solve this problem, because the FROM clause is executed before the GROUP BY clause, as shown in Example 6.76.

**EXAMPLE 6.76**

```
USE sample;
SELECT enter_month
FROM (SELECT MONTH(enter_date) as enter_month
       FROM works_on) AS m
GROUP BY enter_month;
```
The result is

<table>
<thead>
<tr>
<th>enter_month</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

Generally, it is possible to write a table expression any place in a SELECT statement where a table can appear. (The result of a table expression is always a table or, in a special case, an expression.) Example 6.77 shows the use of a table expression in a SELECT list.

**EXAMPLE 6.77**

```
USE sample;
SELECT w.job, (SELECT e.emp_lname
               FROM employee e WHERE e.emp_no = w.emp_no) AS name
FROM works_on w
WHERE w.job IN('Manager', 'Analyst');
```

The result is

<table>
<thead>
<tr>
<th>job</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyst</td>
<td>Jones</td>
</tr>
<tr>
<td>Manager</td>
<td>Jones</td>
</tr>
<tr>
<td>Analyst</td>
<td>Hansel</td>
</tr>
<tr>
<td>Manager</td>
<td>Bertoni</td>
</tr>
</tbody>
</table>

**Common Table Expressions**

A common table expression (CTE) is a named table expression that is supported by Transact-SQL. There are two types of queries that use CTE:

- Nonrecursive queries
- Recursive queries
The following sections describe both query types.

**NOTE**

Common table expressions are also used by the APPLY operator, which allows you to invoke a table-valued function for each row returned by an outer table expression of a query. This operator is discussed in Chapter 8.

**CTEs and Nonrecursive Queries**

The nonrecursive form of a CTE can be used as an alternative to derived tables and views. Generally, a CTE is defined using the WITH statement and an additional query that refers to the name used in WITH (see Example 6.79).

**NOTE**

The WITH keyword is ambiguous in the Transact-SQL language. To avoid ambiguity, you have to use a semicolon (;) to terminate the statement preceding the WITH statement.

Examples 6.78 and 6.79 use the *AdventureWorks* database to show how CTEs can be used in nonrecursive queries. Example 6.78 uses the “convenient” features, while Example 6.79 solves the same problem using a nonrecursive query.

**EXAMPLE 6.78**

```sql
USE AdventureWorks;
SELECT SalesOrderID
FROM Sales.SalesOrderHeader
WHERE TotalDue > (SELECT AVG(TotalDue)
                     FROM Sales.SalesOrderHeader
                      WHERE YEAR(OrderDate) = '2002')
    AND Freight > (SELECT AVG(TotalDue)
                    FROM Sales.SalesOrderHeader
                     WHERE YEAR(OrderDate) = '2002')/2.5;
```

The query in Example 6.78 finds total dues whose values are greater than the average of all dues and whose freights are greater than 40 percent of the average of all dues. The main property of this query is that it is space-consuming, because an inner query has to be written twice. One way to shorten the syntax of the query is to create a view containing the inner query, but that is rather complicated because you would have to create the view and then drop it when you are done with the query. A better way is to write a CTE. Example 6.79 shows the use of the nonrecursive CTE, which shortens the definition of the query in Example 6.78.
**EXAMPLE 6.79**

```sql
USE AdventureWorks;
WITH price_calc(year_2002) AS
  (SELECT AVG(TotalDue)
     FROM Sales.SalesOrderHeader
     WHERE YEAR(OrderDate) = '2002')
SELECT SalesOrderID
FROM Sales.SalesOrderHeader
    WHERE TotalDue > (SELECT year_2002 FROM price_calc)
    AND Freight > (SELECT year_2002 FROM price_calc)/2.5;
```

The syntax for the WITH clause in nonrecursive queries is

```sql
WITH cte_name (column_list) AS
  (inner_query)
outer_query
```

*cte_name* is the name of the CTE that specifies a resulting table. The list of columns that belong to the table expression is written in brackets. (The CTE in Example 6.79 is called *price_calc* and has one column, *year_2002*.) *inner_query* in the CTE syntax defines the SELECT statement, which specifies the result set of the corresponding table expression. After that, you can use the defined table expression in an outer query. (The outer query in Example 6.79 uses the CTE called *price_calc* and its column *year_2002* to simplify the inner query, which appears twice.)

**CTEs and Recursive Queries**

**NOTE**

The material in this subsection is complex. Therefore, you might want to skip it on the first reading of the book and make a note to yourself to return to it.

You can use CTEs to implement recursion because CTEs can contain references to themselves. The basic syntax for a CTE for recursive queries is

```sql
WITH cte_name (column_list) AS
  (anchor_member
    UNION ALL
    recursive_member)
outer_query
```
**cte_name** and **column_list** have the same meaning as in CTEs for nonrecursive queries. The body of the WITH clause comprises two queries that are connected with the UNION ALL operator. The first query will be invoked only once, and it starts to accumulate the result of the recursion. The first operand of UNION ALL does not reference the CTE (see Example 6.80). This query is called the **anchor query** or **seed**.

The second query contains a reference to the CTE and represents the recursive portion of it. For this reason it is called the **recursive member**. In the first invocation of the recursive part, the reference to the CTE represents the result of the anchor query. The recursive member uses the query result of the first invocation. After that, the system repeatedly invokes the recursive part. The invocation of the recursive member ends when the result of the previous invocation is an empty set.

The UNION ALL operator joins the rows accumulated so far, as well as the additional rows that are added in the current invocation. (Inclusion of UNION ALL means that no duplicate rows will be eliminated from the result.)

Finally, **outer query** defines a query specification that uses the CTE to retrieve all invocations of the union of both members.

The table definition in Example 6.80 will be used to demonstrate the recursive form of CTEs.

**EXAMPLE 6.80**

```sql
USE sample;
CREATE TABLE airplane
    (containing_assembly VARCHAR(10),
    contained_assembly VARCHAR(10),
    quantity_contained INT,
    unit_cost DECIMAL (6,2));
insert into airplane values ( 'Airplane', 'Fuselage',1, 10);
insert into airplane values ( 'Airplane', 'Wings', 1, 11);
insert into airplane values ( 'Airplane', 'Tail',1, 12);
insert into airplane values ( 'Fuselage', 'Cockpit', 1, 13);
insert into airplane values ( 'Fuselage', 'Cabin', 1, 14);
insert into airplane values ( 'Fuselage', 'Nose',1, 15);
insert into airplane values ( 'Cockpit', NULL, 1,13);
insert into airplane values ( 'Cabin', NULL, 1, 14);
insert into airplane values ( 'Nose', NULL, 1, 15);
insert into airplane values ( 'Wings', NULL,2, 11);
insert into airplane values ( 'Tail', NULL, 1, 12);
```

The **airplane** table contains four columns. The column **containing_assembly** specifies an assembly, while **contained_assembly** comprises the parts (one by one) that build the corresponding assembly. (Figure 6-1 shows graphically how an airplane with its parts could look.)
Suppose that the `airplane` table contains 11 rows, which are shown in Table 6-2. (The INSERT statements in Example 6.80 insert these rows in the `airplane` table.)

Example 6.81 shows the use of the WITH clause to define a query that calculates the total costs of each assembly.

**EXAMPLE 6.81**

```sql
USE sample;
WITH list_of_parts(assembly1, quantity, cost) AS
    (SELECT containing_assembly, quantity_contained, unit_cost
     FROM airplane
     WHERE contained_assembly IS NULL
     UNION ALL
     SELECT a.containing_assembly, a.quantity_contained,
```
CAST(l.quantity*l.cost AS DECIMAL(6,2))
FROM list_of_parts l,airplane a
WHERE l.assembly1 = a.contained_assembly)

SELECT * FROM list_of_parts;

The WITH clause defines the CTE called list_of_parts, which contains three
columns: assembly, quantity, and cost. The first SELECT statement in Example 6.81
will be invoked only once, to accumulate the results of the first step in the recursion
process.

The SELECT statement in the last row of Example 6.81 displays the following
result:

<table>
<thead>
<tr>
<th>assembly</th>
<th>quantity</th>
<th>costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cockpit</td>
<td>1</td>
<td>13.00</td>
</tr>
<tr>
<td>Cabin</td>
<td>1</td>
<td>14.00</td>
</tr>
<tr>
<td>Nose</td>
<td>1</td>
<td>16.50</td>
</tr>
<tr>
<td>Wings</td>
<td>2</td>
<td>11.00</td>
</tr>
<tr>
<td>Tail</td>
<td>1</td>
<td>12.00</td>
</tr>
<tr>
<td>Airplane</td>
<td>1</td>
<td>12.00</td>
</tr>
<tr>
<td>Airplane</td>
<td>1</td>
<td>22.00</td>
</tr>
<tr>
<td>Fuselage</td>
<td>1</td>
<td>16.50</td>
</tr>
<tr>
<td>Airplane</td>
<td>1</td>
<td>16.50</td>
</tr>
<tr>
<td>Fuselage</td>
<td>1</td>
<td>14.00</td>
</tr>
<tr>
<td>Airplane</td>
<td>1</td>
<td>14.00</td>
</tr>
<tr>
<td>Fuselage</td>
<td>1</td>
<td>13.00</td>
</tr>
<tr>
<td>Airplane</td>
<td>1</td>
<td>13.00</td>
</tr>
</tbody>
</table>

The first five rows in the preceding output show the result set of the first invocation
of the anchor member of the query in Example 6.81. All other rows are the result of
the recursive member (second part) of the query in the same example. The recursive
member of the query will be invoked twice: the first time for the fuselage assembly and
the second time for the airplane itself.

The query in Example 6.82 will be used to get the costs for each assembly with all its
subparts.

**EXAMPLE 6.82**

USE sample;
WITH list_of_parts(assembly, quantity, cost) AS
  (SELECT containing_assembly, quantity_contained, unit_cost
The output of the query in Example 6.82 is as follows:

<table>
<thead>
<tr>
<th>assembly</th>
<th>parts</th>
<th>sum_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airplane</td>
<td>5</td>
<td>76.00</td>
</tr>
<tr>
<td>Cabin</td>
<td>1</td>
<td>14.00</td>
</tr>
<tr>
<td>Cockpit</td>
<td>1</td>
<td>13.00</td>
</tr>
<tr>
<td>Fuselage</td>
<td>3</td>
<td>42.00</td>
</tr>
<tr>
<td>Nose</td>
<td>1</td>
<td>16.500</td>
</tr>
<tr>
<td>Tail</td>
<td>1</td>
<td>12.00</td>
</tr>
<tr>
<td>Wings</td>
<td>2</td>
<td>11.00</td>
</tr>
</tbody>
</table>

There are several restrictions for a CTE in a recursive query:

- The CTE definition must contain at least two SELECT statements (an anchor member and one recursive member) combined by the UNION ALL operator.
- The number of columns in the anchor and recursive members must be the same. (This is the direct consequence of using the UNION ALL operator.)
- The data type of a column in the recursive member must be the same as the data type of the corresponding column in the anchor member.
- The FROM clause of the recursive member must refer only once to the name of the CTE.
- The following options are not allowed in the definition part of a recursive member: SELECT DISTINCT, GROUP BY, HAVING, aggregation functions, TOP, and subqueries. (Also, the only join operation that is allowed in the query definition is an inner join.)
Summary

This chapter covered all the features of the SELECT statement regarding data retrieval from one or more tables. Every SELECT statement that retrieves data from a table must contain at least a SELECT list and the FROM clause. The FROM clause specifies the table(s) from which the data is retrieved. The most important optional clause is the WHERE clause, containing one or more conditions that can be combined using the Boolean operators AND, OR, and NOT. Hence, the conditions in the WHERE clause place the restriction on the selected row.

Exercises

E.6.1
Get all rows of the works_on table.

E.6.2
Get the employee numbers for all clerks.

E.6.3
Get the employee numbers for employees working on project p2 and having employee numbers lower than 10000. Solve this problem with two different but equivalent SELECT statements.

E.6.4
Get the employee numbers for employees who didn't enter their project in 2007.

E.6.5
Get the employee numbers for all employees who have a leading job (i.e., Analyst or Manager) in project p1.

E.6.6
Get the enter dates for all employees in project p2 whose jobs have not been determined yet.

E.6.7
Get the employee numbers and last names of all employees whose first names contain two letter t’s.
Get the employee numbers and first names of all employees whose last names have a letter o or a as the second character and end with the letters es.

Find the employee numbers of all employees whose departments are located in Seattle.

Find the last and first names of all employees who entered their projects on 04.01.2007.

Group all departments using their locations.

What is a difference between the DISTINCT and GROUP BY clauses?

How does the GROUP BY clause manage the NULL values? Does it correspond to the general treatment of these values?

What is the difference between COUNT(*) and COUNT(column)?

Find the highest employee number.

Get the jobs that are done by more than two employees.

Find the employee numbers of all employees who are clerks or work for department d3.

Why is the following statement wrong?

```
SELECT project_name
    FROM project
    WHERE project_no =
        (SELECT project_no FROM works_on WHERE Job = 'Clerk')
```
Write the correct syntax form for the statement.

E.6.19
What is a practical use of temporary tables?

E.6.20
What is a difference between global and local temporary tables?

NOTE
Write all solutions for the following exercises that use a join operation using the explicit join syntax.

E.6.21
For the project and works_on tables, create the following:
   a. Natural join
   b. Cartesian product

E.6.22
If you intend to join several tables in a query (say n tables), how many join conditions are needed?

E.6.23
Get the employee numbers and job titles of all employees working on project Gemini.

E.6.24
Get the first and last names of all employees who work for department Research or Accounting.

E.6.25
Get the enter dates of all clerks who belong to the department d1.

E.6.26
Get the names of projects on which two or more clerks are working.

E.6.27
Get the first and last names of the employees who are managers and work on project Mercury.
E.6.28
Get the first and last names of all employees who entered the project at the same time as at least one other employee.

E.6.29
Get the employee numbers of the employees living in the same location and belonging to the same department as one another. (Hint: Use the extended sample database.)

E.6.30
Get the employee numbers of all employees belonging to the Marketing department. Find two equivalent solutions using

- The join operator
- The correlated subquery