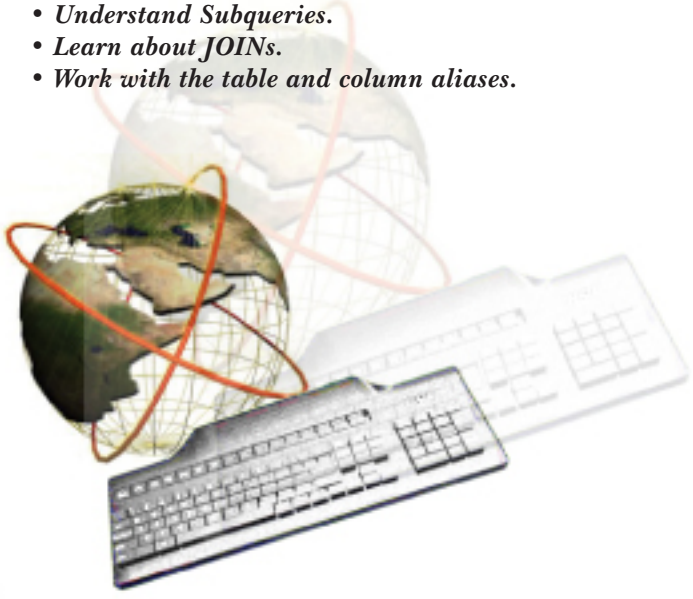
Chapter 3

Retrieving Data in SQL

Learning Objectives

After completing this chapter, you will be able to:

- *Use the SELECT statement.*
- *Work with SQL operators.*
- *Understand the Set operators (UNION, UNION ALL, INTERSECT, and MINUS).*
- *Understand the operator precedence.*
- *Understand Subqueries.*
- *Learn about JOINs.*
- *Work with the table and column aliases.*



THE SELECT STATEMENT

The **SELECT** statement is the most popular SQL statement used for querying a table. This statement is used to retrieve or view the data of one or more tables. The syntax for using the **SELECT** statement is as follows:

```
SELECT *
FROM Table_Name;
```

In the above syntax, **SELECT** and **FROM** are keywords and **Table_Name** is the name of the table from which you want to view data rows. ***** (asterisk) is also a keyword and is used to retrieve data from all columns or fields of a table.

For example:

Enter the following query in SQL *Plus to get employee records, as shown in Figure 3-1.

```
SELECT * FROM Emp;
```

Figure 3-1 shows the output of the above query when you execute it.

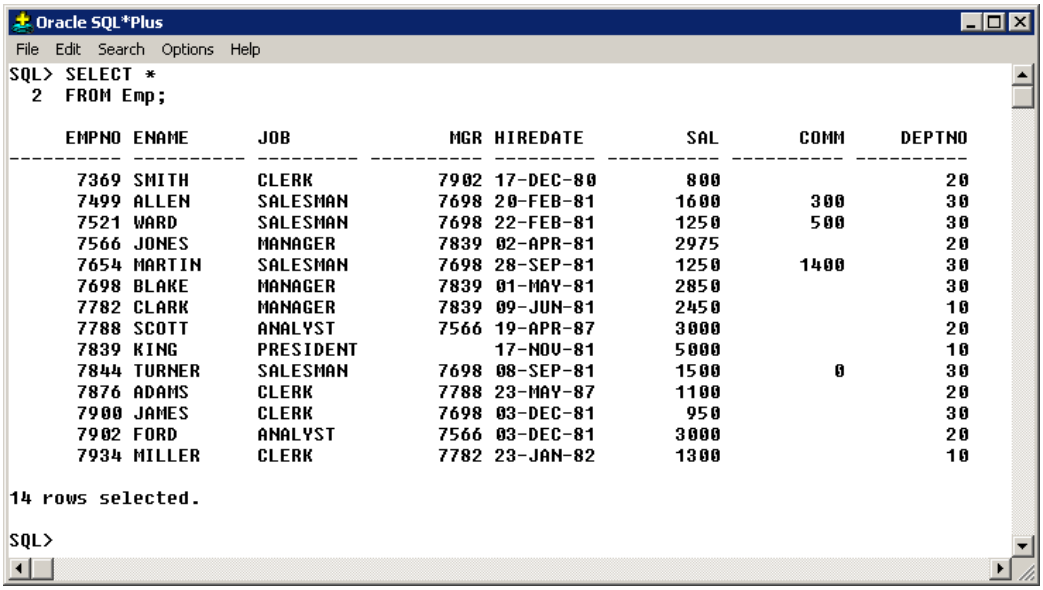


Figure 3-1 *The rows retrieved from the* ***Emp*** *table*

The above SQL query retrieves all information contained within the **Emp** table. Note that the asterisk is used as a wildcard in SQL. Literally, it means "Select all records from the a table."

You can use the following syntax to limit the attributes retrieved from a table:

```
SELECT Column1, Column2, ...
FROM Table_Name;
```

In the above syntax, **SELECT** and **FROM** are keywords and **Column1**, **Column2**, ... are the names of the columns from which you want to retrieve data. **Table_Name** is name of the table from which you want to retrieve data.

For example:

The Human Resources department may require a list of names of all employees of a company. You can retrieve the required information using the following SQL statement:

```
SELECT Ename FROM Emp;
```

Selecting Distinct Rows

You can retrieve distinct rows from a table by using the **DISTINCT** clause with the **SELECT** statement. Retrieving distinct rows from the table prevents the selection of duplicate rows. Following is the syntax for using the **DISTINCT** clause with the **SELECT** statement:

```
SELECT DISTINCT Column_Name
FROM Table_Name;
```

In the above syntax, **Column_Name** is the name of the column from which you want to retrieve distinct values and **Table_Name** is the name of the table which contains the column **Column_Name**.



Note

*You can also use the **UNIQUE** keyword instead of the **DISTINCT** keyword to prevent the selection of duplicate rows.*

For example:

Enter the following query in SQL *Plus and then execute it. The output of the query will be displayed, as shown in Figure 3-2.

```
SELECT Job FROM Emp;
```

This query will retrieve a list of jobs. Notice that the jobs **ANALYST**, **CLERK**, **MANAGER**, and **SALESMAN** appear more than once. Now, if you want to retrieve the list of different jobs with no job being repeated in the list, use the **DISTINCT** clause with the **SELECT** statement, as shown below. Figure 3-3 shows the output of the following query:

```
SELECT DISTINCT Job FROM Emp;
```

The above query retrieves all distinct job names from the **Emp** table.

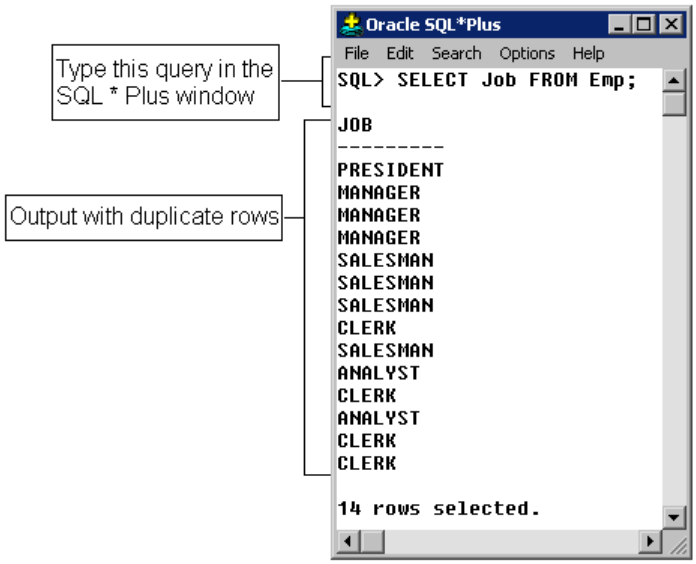


Figure 3-2 *Output with duplicate rows*

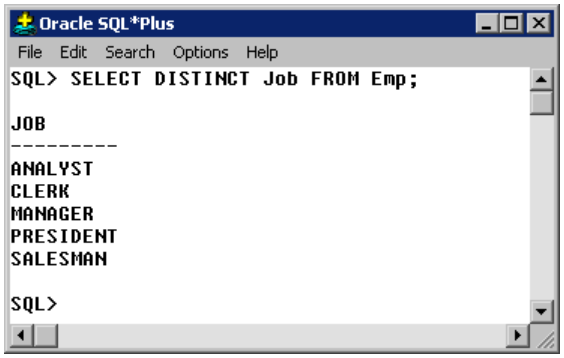


Figure 3-3 *Output with distinct rows*

Selecting Rows with the WHERE Clause

The **WHERE** clause is used with the **SELECT**, **DELETE**, or **UPDATE** statement to select, delete, or update the data from a table on the basis of a condition. Also, this clause is used to filter the data from the database. The **WHERE** clause selects, deletes, or updates only those rows in which expressions evaluate to true. The syntax for using the **WHERE** clause is as follows:

```
SELECT Column_Name
FROM Table_Name
WHERE Column_Name/Expression Operator Value/Expression;
```

In the above syntax, **SELECT**, **FROM**, and **WHERE** are the keywords; **Column_Name** is the name of the column that you want to select from the table; and **Table_Name** is the name of

the table. The **WHERE** clause used with both the **DELETE** and **UPDATE** statements will be discussed later in this chapter.

For example:

Enter the following command lines in SQL *Plus and then execute them. The output of the query will be displayed, as shown in Figure 3-4.

```
SQL> SELECT Empno, Ename "Name", HireDate "Join Date",  Sal "Salary", Job
FROM Emp WHERE Job= 'MANAGER';
```

In the above SQL statement, the **WHERE** clause filters the results set from the **Emp** table. The above SQL statement will return all rows having the job Manager from the **Emp** table.

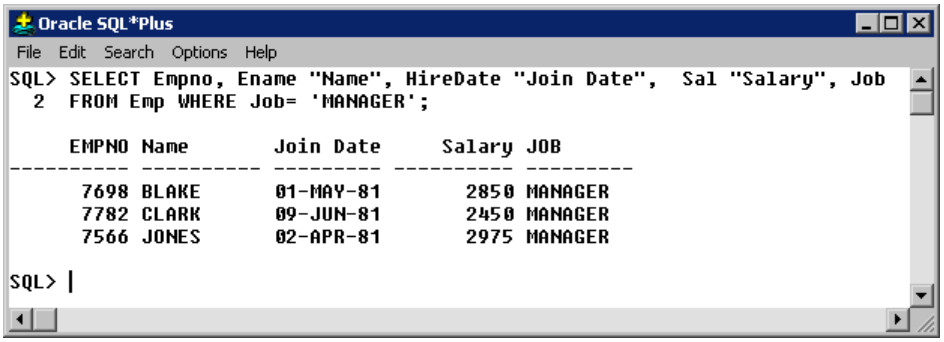


Figure 3-4 *The **SELECT** statement with the **WHERE** clause*



Note

*The column alias used in the above query such as **Name**, **Join Date**, and **Salary** will be discussed later in this chapter*

SQL OPERATORS

The following operators in Oracle are supported by SQL *Plus:

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Other operators
5. Set Operators

These operators are discussed next.

Arithmetic Operators

The Oracle database uses arithmetic expressions in SQL commands to perform calculations, based on numeric values. An arithmetic expression consists of column names connected with

the number data type through an arithmetic operator. The arithmetic operators and their usage are discussed in Table 3-1.

Operator	Description
+, -	These are unary operators that represent the positive and negative expressions.
+	This is the addition operator. It is used to add two data items or expressions. It is a binary operator.
-	This is the subtraction operator. It is used to subtract two data items or expressions. It is also a binary operator.
*	This is the multiplication operator. It is used to multiply two data items or expressions. It is also a binary operator.
/	This is the division operator. It is used to divide two data items or expressions. It is also a binary operator.

Table 3-1 *The Arithmetic operators and their description*

Following are the examples of arithmetic operators.

The following query is used to add two values in Oracle:

```
SELECT 5 + 5 Total_Value FROM
DUAL;

TOTAL_VALUE
--------------------
10
```

The result of the above query will be stored in the **TOTAL_VALUE** column of the numeric data type.

The following query is used to divide a value with other value:

```
SELECT 8 / 2 Total_Value FROM
DUAL;

TOTAL_VALUE
--------------------
4
```

The following query adds specified number of days to SYSDATE:

```
SELECT SYSDATE, (SYSDATE) + 10
result_date FROM dual;

SYSDATE        RESULT_DATE
-------------  ---------------------
25-JUN-08      05-JUL-08
```

Comparison Operators

The Comparison operators are used to compare one expression with another expression. These operators compare two values or expressions and return a boolean result TRUE, FALSE, or NULL. The Comparison operators are **=** (Equal), **<** (Less than), **>** (Greater than), **<=** (Less than or equal to), **>=** (Greater than or equal to), **<>** and **!=** (Not equal to), and value comparisons. These operators are discussed next.

= (Equal)

This operator is used in a conditional statement. If the values or the result of expression on both sides of the operator are equal, the condition will be TRUE.

For example:

```
SELECT Ename, Sal, Deptno FROM Emp
WHERE Deptno = 30;
```

In the above example, the SQL query will return all those records of the **Emp** table in which the department number is 30. The above query will return the following rows:

```
ENAME      SAL        DEPTNO
---------- ---------- ----------
BLAKE      2850       30
MARTIN     1250       30
ALLEN      1600       30
TURNER     1500       30
JAMES      950        30
WARD       1250       30
```

!=, <>, or ^ = (Not Equal to)

These operators are used to check inequality. If the value or result of expression on both sides of the operator is not equal, the condition will evaluate to TRUE.

For example:

```
SELECT Ename, Sal, Deptno FROM Emp
WHERE Deptno <> 30;
```

In the above example, the SQL query will return all those records of the **Emp** table, in which the department number is not 30. The above query returns the following rows:

```
ENAME      SAL        DEPTNO
---------- ---------- ----------
KING       5000       10
CLARK      2450       10
JONES      2975       20
FORD       3000       20
SMITH      800        20
SCOTT      3000       20
ADAMS      1100       20
MILLER     1300       10
```

< (Less Than)

If the value or result of an expression on the left of the operator is less than the value or result of an expression on the right side of the operator, the **<** (Less than) operator will evaluate to TRUE.

For example:

```
SELECT Ename, Sal, Deptno FROM Emp
WHERE Sal < 2000;
```

In the above example, the SQL query will return all those records from the **Emp** table in which the salary is less than **2000**. The above query will return the following rows:

```
ENAME      SAL        DEPTNO
---------- ---------- ----------
MARTIN     1250       30
ALLEN      1600       30
TURNER     1500       30
JAMES      950        30
WARD       1250       30
SMITH      800        20
ADAMS      1100       20
MILLER     1300       10
```

> (Greater Than)

If the value or the result of an expression on the left of the operator is more than the value or result of an expression on the right, the **>** (Greater than) operator will evaluate to TRUE.

For example:

```
SELECT Ename, Sal, Deptno FROM Emp
WHERE Sal > 2000;
```

In the above example, the SQL query will return all those records from the **Emp** table in which the salary is greater than **2000**. The above query will return the following rows:

ENAME	SAL	DEPTNO
KING	5000	10
BLAKE	2850	30
CLARK	2450	10
JONES	2975	20
FORD	3000	20
SCOTT	3000	20

<= (Less Than or Equal to)

If the value or result of an expression on the left of this operator is either less than or equal to the value or result of an expression on the right of the operator, the **<=** (Less then or Equal to) operator will evaluate to TRUE.

For example:

```
SELECT Empno, Ename, Sal, Deptno FROM Emp
WHERE Sal <= 3000;
```

In the above example, the SQL query returns all those records of the **Emp** table in which the salary is less than or equal to **3000**. The above query will return the following rows:

EMPNO	ENAME	SAL	DEPTNO
7698	BLAKE	2850	30
7782	CLARK	2450	10
7566	JONES	2975	20
7654	MARTIN	1250	30
7499	ALLEN	1600	30
7844	TURNER	1500	30
7900	JAMES	950	30
7521	WARD	1250	30
7902	FORD	3000	20
7369	SMITH	800	20
7788	SCOTT	3000	20
7876	ADAMS	1100	20
7934	MILLER	1300	10

>= (Greater Than or Equal To)

If the value or result of an expression on the left of this operator is either greater than or equal to the value or expression on the right of this operator, the **>=** (Greater than or Equal to) operator will evaluate to TRUE

For example:

```
SELECT Empno, Ename, Sal, Deptno FROM Emp
WHERE Sal >= 3000;
```

In the above example, the SQL query will return all those records of the **Emp** table in which the salary is either equal to or more than **3000**. The above query will return the following rows:

EMPNO	ENAME	SAL	DEPTNO
7839	KING	5000	10
7902	FORD	3000	20
7788	SCOTT	3000	20

ANY or SOME

The **ANY** or **SOME** operator is used to compare a value with each value in a list or the values returned by a query. These operators must be preceded by a Comparison operator **=**, **!=**, **>**, **<**, **<=**, or **>=**.

The following example will illustrate the use of **ANY** and **SOME** operators with the comparison operator **>**(greater than):

```
SELECT Ename, Sal, Job, Hiredate
FROM Emp
WHERE Sal > SOME(1000, 2000, 3000);

Or

SELECT Ename, Sal, Job, Hiredate
FROM Emp
WHERE Sal > ANY(1000, 2000, 3000);
```

In this example, the SQL query will return all those records of the **Emp** table in which the salary of employee is greater than the values in the list (1000, 2000, 3000). The above query will return the following rows:

ENAME	SAL	JOB	HIREDATE
KING	5000	PRESIDENT	17-NOV-81
BLAKE	2850	MANAGER	01-MAY-81
CLARK	2450	MANAGER	09-JUN-81
JONES	2975	MANAGER	02-APR-81
ALLEN	1600	SALESMAN	20-FEB-81
FORD	3000	ANALYST	03-DEC-81
SCOTT	3000	ANALYST	09-DEC-82

The following example will illustrate the use of the **ANY** and **SOME** operators with the comparison operator **=**(Equal):

```
SELECT Ename, Sal, Job, Hiredate
FROM Emp
WHERE Sal = ANY(800, 3000, 5000);
```

Or

```
SELECT Ename, Sal, Job, Hiredate
FROM Emp
WHERE Sal = SOME(800, 3000, 5000);
```

In the above example, the SQL query will return all those records of the **Emp** table in which the employee number is equal to the values in the list (800, 3000, 5000). The above query returns the following rows:

ENAME	SAL	JOB	HIREDATE
KING	5000	PRESIDENT	17-NOV-81
FORD	3000	ANALYST	03-DEC-81
SMITH	800	CLERK	17-DEC-80
SCOTT	3000	ANALYST	09-DEC-82



Note

*When the **ANY** operator is used with the comparison operator **=**(Equal), it works the same way as the **IN** operator. The **IN** operator will be discussed later in this chapter.*

ALL

The **ALL** operator is used to compare a value with every value in a list or the value returned by a query. This operator must be preceded by the comparison operator **=**, **!=**, **>**, **<**, **<=**, or **>=**.

The following example will illustrate the use of the **ALL** operator with the comparison operator **>** (greater than):

```
SELECT Ename, Sal, Job, Hiredate
FROM Emp
WHERE Sal > ALL(500, 1000, 2000);
```

In this example, the SQL query will return all those records of the **Emp** table in which the salary of employee is greater than the values in the list (500, 1000, 2000). The above query will return the following rows:

ENAME	SAL	JOB	HIREDATE
KING	5000	PRESIDENT	17-NOV-81
BLAKE	2850	MANAGER	01-MAY-81
CLARK	2450	MANAGER	09-JUN-81
JONES	2975	MANAGER	02-APR-81
FORD	3000	ANALYST	03-DEC-81
SCOTT	3000	ANALYST	09-DEC-82

The following example will illustrate the use of the **ALL** operator with the comparison operator **>=** (greater than or equal to):

```
SELECT Ename, Sal, Job, Hiredate
FROM Emp
WHERE Sal >= ALL(800, 1000, 2000);
```

In the above example, the SQL query will return all those records of the **Emp** table in which the salary of employee is greater than or equal to the values in the list (800, 900, 1000). The above query will return the following rows:

ENAME	SAL	JOB	HIREDATE
MARKING	5000	PRESIDENT	17-NOV-81
BLAKE	2850	MANAGER	01-MAY-81
CLARK	2450	MANAGER	09-JUN-81
JONES	2975	MANAGER	02-APR-81
MARTIN	1250	SALESMAN	28-SEP-81
ALLEN	1600	SALESMAN	20-FEB-81
TURNER	1500	SALESMAN	08-SEP-81
WARD	1250	SALESMAN	22-FEB-81
FORD	3000	ANALYST	03-DEC-81
SCOTT	3000	ANALYST	09-DEC-82
ADAMS	1100	CLERK	12-JAN-83
MILLER	1300	CLERK	23-JAN-82

Logical Operators

The logical operators are used to combine the results of two or more conditions to produce a single result. The logical operators are discussed next.

NOT

The **NOT** operator is used to reverse the output of any other logical operator. This operator will return TRUE, if the given condition is FALSE, and will return FALSE, if the given condition is TRUE.

For example:

```
SELECT *  FROM Emp
WHERE NOT (Job IS NULL);
```

The above query will return all those records of the **Emp** table in which the column **Job** is not Null.

The following example will illustrate the use of the **BETWEEN** operator with the **NOT** operator.

```
SELECT * FROM Emp
WHERE NOT (Sal BETWEEN 1000 AND 2000);
```

The above query will return all those records of the **Emp** table in which the salary is not between 1000 and 2000.

AND

The **AND** operator joins two or more than two conditions. This operator will return TRUE, if both conditions are TRUE, and will return FALSE, if one of the conditions is FALSE. Otherwise, it will return an unknown value.

For example:

```
SELECT *  FROM Emp
WHERE  Job = 'CLERK' AND Sal > 900;
```

The above query will return all those records of the **Emp** table in which both the conditions, **Job = 'CLERK'** and **Sal > 900**, return TRUE, as shown in Figure 3-5.

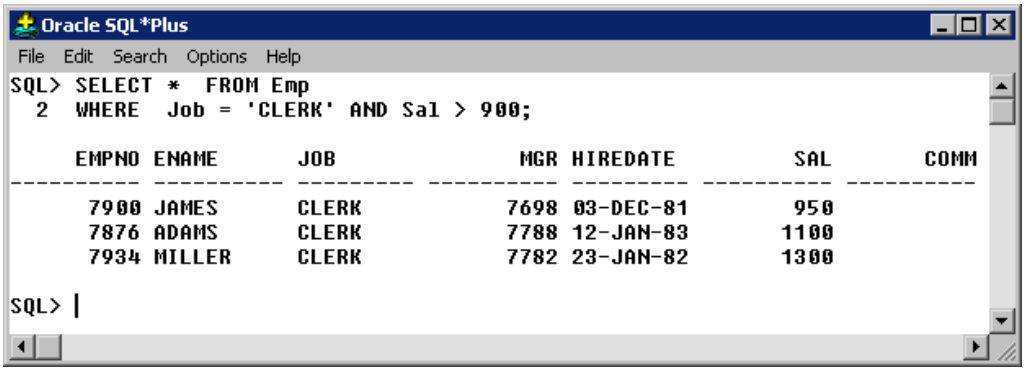


Figure 3-5 *Query showing the use of the* ***AND*** *operator*

OR

The **OR** operator joins two or more than two conditions. This operator will return TRUE, if one of the conditions evaluates to TRUE and return FALSE, if both the conditions evaluate to FALSE. Otherwise, it will return an unknown value. The **OR** operator is evaluated after the **AND** operator.

```
SELECT *  FROM Emp
WHERE  Job = 'CLERK' OR Sal > 2000;
```

This above query will return all those records from the **Emp** table in which one of the conditions, **Job = 'CLERK'** or **Sal > 2000**, is TRUE, as shown in Figure 3-6.

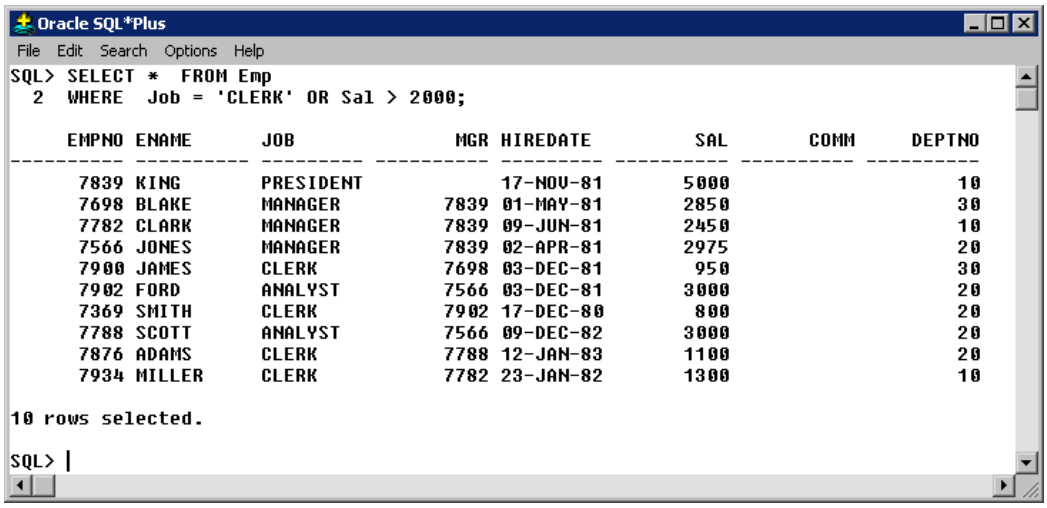


Figure 3-6 *Query showing the use of the* ***OR*** *operator*

Other Operators

Oracle provides some other operators as well. These are discussed next.

LIKE Operator

You can use the **LIKE** operator in a character string. This operator compares the string with the matching pattern. Sometimes, you may need to perform searches by matching part of a character string. In such cases, you can use the **LIKE** operator. For example, you may need to retrieve the name of the students, whose last name begins with the letter M, or find all courses with the initial letters MIS. To do so, you can use the **LIKE** operator. The general syntax for using the **LIKE** operator in the search condition is as follows:

```
SELECT Column1, Column2.....
FROM Table
WHERE Column_Name LIKE 'Char_String';
```

In the above syntax, **Char_String** is the pattern with which the **Column_Name** will be compared. The pattern is a value having the data type **CHAR** or **VARCHAR2** and contains the special matching pattern characters: percent sign (**%**) and underscore (**_**).

The percent sign (**%**) denotes single number or multiple numbers of unknown characters, and underscore sign (**_**) denotes only an unknown character.

For example:

```
SELECT Empno, Ename, Sal, Job, Hiredate
FROM Emp WHERE Ename LIKE 'J%';
```

The above query will return employee number and names whose name starts with letter J. The output of the above query is shown in Figure 3-7.

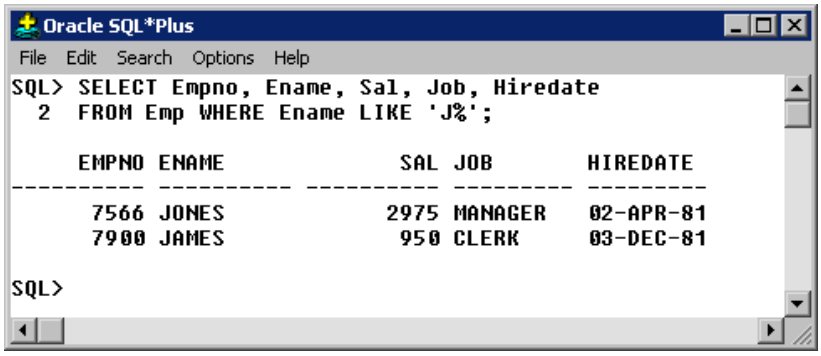


Figure 3-7 *Query showing the use of the (%)* ***Like*** *operator*

The following example will illustrate the use of the **LIKE** operator with the matching pattern characters: percent sign (%) and underscore (_).

Example 1

Write queries that will illustrate the use of the **LIKE** operator with the matching pattern characters percent sign (%) and underscore (_).

The following steps are required to use the **LIKE** operator.

1. In SQL *Plus, enter the following SQL query to retrieve the rows in which the name of employees begins with the letter **A**:

   ```
   SELECT Ename, Sal, Job, Comm, Hiredate FROM Emp
   WHERE Ename LIKE 'A%';
   ```

   In the above example, the percent sign **(%)** used after the character **A** in the **LIKE** operator represents any possible character or a set of characters that may appear after **A**. Thus, the above query will return all those employee names that begin with the character **A**, as shown in Figure 3-8.

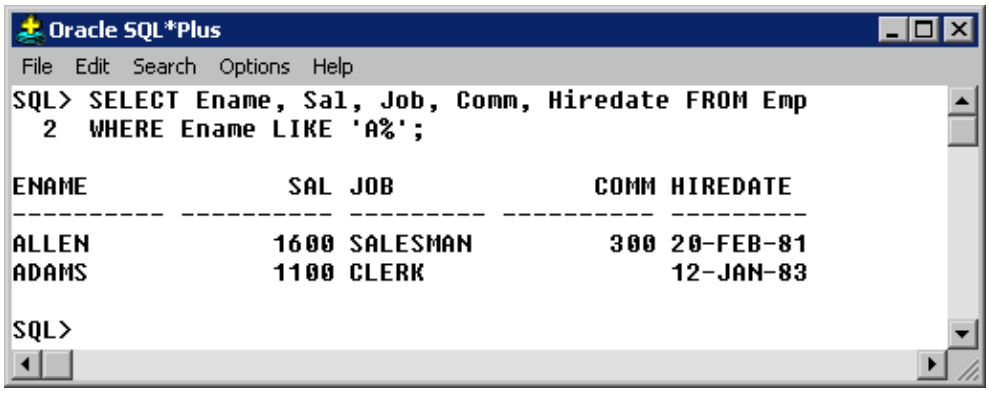


Figure 3-8 *Query showing the use of the* ***(%) LIKE*** *operator*

2. In SQL * Plus, enter the following SQL query to retrieve the rows in which the name of employees contains the word **ES**:

```
SELECT Ename, Sal, Job, Comm, Hiredate
FROM Emp
WHERE Ename LIKE '%ES%';
```

The above query will return name, salary, job, and commission of those employees whose name contains the characters **ES**. Note that this character set may appear anywhere in the name of the employees.

3. In SQL *Plus, enter the following SQL query to retrieve the rows in which the name of employee is similar to JONES and so on. In these names, the first and the last two characters will remain the same. This can be done by using the underscore (_) with the **LIKE** operator.

```
SELECT * FROM Emp
WHERE Ename LIKE 'J__ES';
```

In the above example, the underscore sign (_) used twice between **J** and **ES** in the **LIKE** operator represents any possible two characters that might appear between **J** and **ES**. The output of the above query is shown in Figure 3-9.

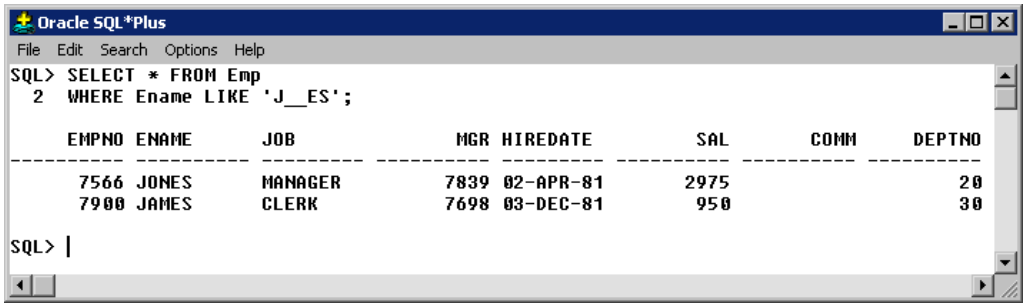


Figure 3-9 *Query showing the use of the (_)* ***LIKE*** *operator*

BETWEEN and NOT BETWEEN Operators

The **BETWEEN** operator is used in the **WHERE** clause to select a range of data between two values or expressions. The syntax for using the **BETWEEN** operator is as follows:

```
SELECT Column1, Column2.....
FROM Table
WHERE Column_Name BETWEEN Value1 AND Value2;
```

In the above syntax, **BETWEEN** is a keyword. **Value1** and **Value2** are the start and end values respectively. Note that the start value **Value1** should always be less than the end value **Value2**.

The above SQL statement will return the records where **Column_Name** is within the range of **Value1** and **Value2**. The **BETWEEN** operator can be used in any valid SQL statement such as **SELECT**, **INSERT**, **UPDATE**, or **DELETE**.

For example:

In SQL * Plus, enter the following SQL query to retrieve the rows from the **Emp** table having employee number between 7521 and 7844:

```
SELECT * FROM Emp
WHERE Empno BETWEEN 7521 AND 7844;
```

The above query will return all details of employees having employee number between 7521 and 7844, as shown in Figure 3-10.

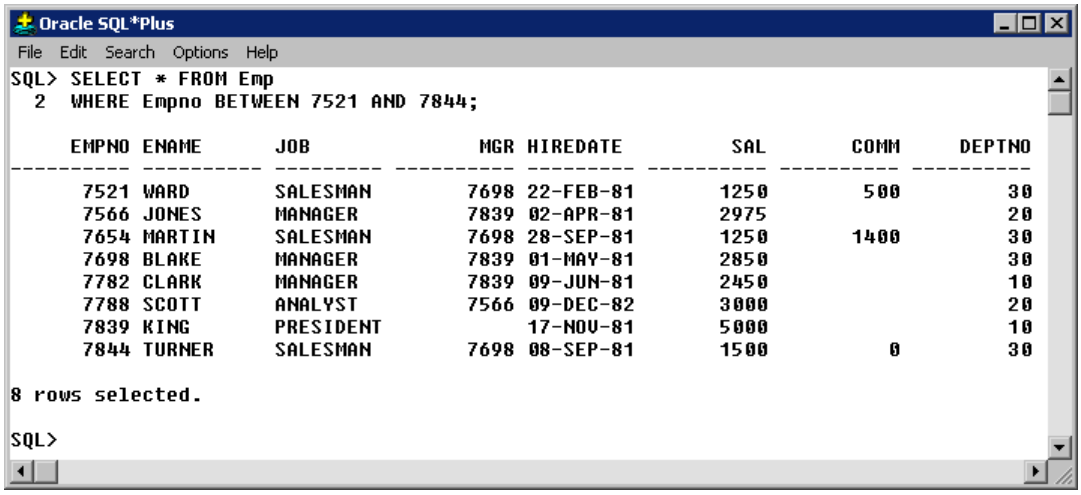


Figure 3-10 *Query showing the use of the* ***BETWEEN*** *operator*

The following example will illustrate the use of the **BETWEEN** operator with the DATE data type:

```
SELECT * FROM Emp
WHERE Hiredate BETWEEN TO_DATE('25/06/1981', 'dd/mm/yy') AND
TO_DATE('25/06/2006', 'dd/mm/yy');
```

The preceding query will return all details of employees having **Hiredate** between Jun 25, 1981 and Jun 25, 2006, as shown in Figure 3-11.

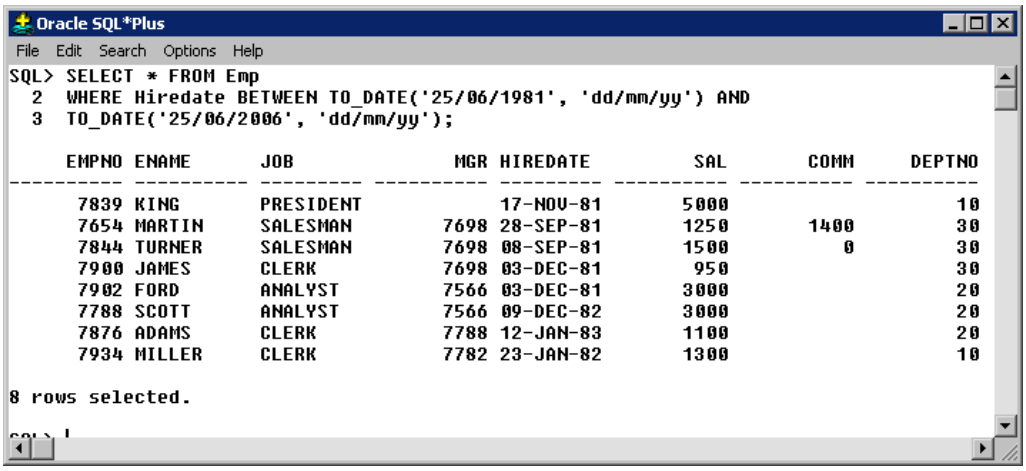


Figure 3-11 *Query showing the use of the* ***BETWEEN*** *operator with the* ***DATE*** *data type*

The above SQL statement is equivalent to the following SQL statement:

```
SELECT * FROM Emp
WHERE Hiredate >= TO_DATE('25/06/1981', 'dd/mm/yy')
AND Hiredate <= TO_DATE('25/06/2006', 'dd/mm/yy');
```

NOT BETWEEN

You can combine the **BETWEEN** operator with the **NOT** operator. The **NOT BETWEEN** operator is used to select a range of data that does not exists between the two given values or expressions.

For example:

```
SELECT * FROM Emp
WHERE Empno NOT BETWEEN 7521 AND 7844;
```

The above query will return all details of those employees whose **Empno** is not between 7521 and 7844, as shown in Figure 3-12.

The above query can be also written as:

```
SELECT * FROM Emp
WHERE Empno < 7521 OR Empno > 7844;
```

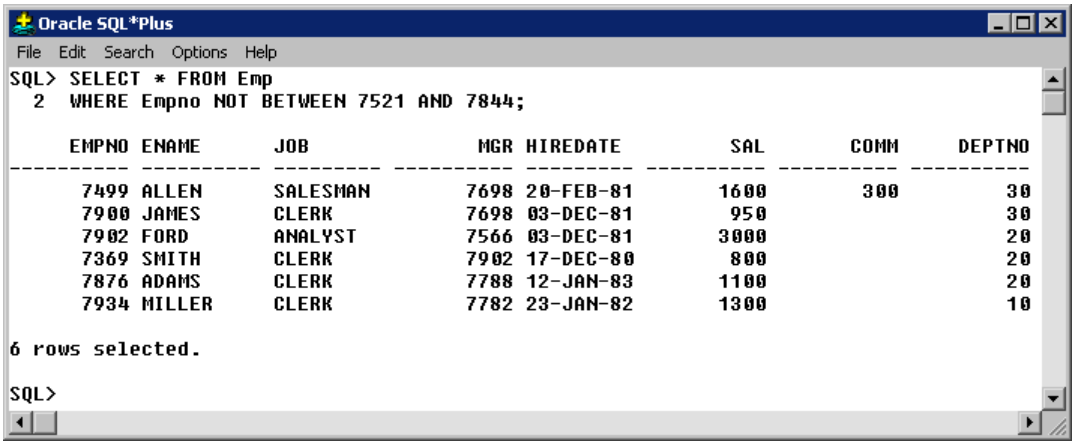


Figure 3-12 *Query showing the use of the* ***NOT BETWEEN*** *operator*

IN and NOT IN Operators

The **IN** operator is used to compare a value with each value in a list or returned by a query. The syntax for using the **IN** operator is as follows:

```
SELECT Column1, Column2.....
FROM Table
WHERE Column_Name IN (Value1, Value2, Value3,..... Value_n| Select_statement);
```

The above SQL statement will return all those records in which **Column_Name** is **Value1**, **Value2**, **Value3**, ... **Value_n**. The values in the parenthesis can be one or more, with each value separated by a comma. The values can be characters or numerical. The **IN** operator can be used with any valid SQL statement: **SELECT**, **INSERT**, **UPDATE**, or **DELETE**.

For example:

In SQL *Plus, enter the following query to retrieve details of those employees whose employee numbers are 7499, 7900, and 7902.

```
SELECT * FROM Emp
WHERE Empno IN(7499, 7900, 7902);
```

The list of values enclosed in the parenthesis is called an inlist. The above query has an inlist with three values (7499, 7900, 7902). The above query will return the details of those employees whose employee number is same as in the inlist, as shown in Figure 3-13.

The following example will illustrate the use of the **IN** operator with string values in the inlist.

```
SELECT * FROM Emp
WHERE Job IN ('CLERK', 'MANAGER');
```

The above query will list the names of all employees having **CLERK** and **MANAGER** as their **Job**, as shown in Figure 3-14. In each of these queries, the **IN** operator has been used to select the data based on multiple constant values.

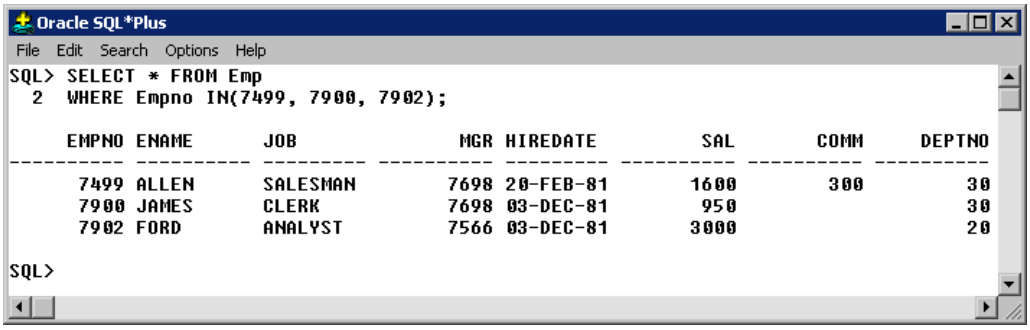


Figure 3-13 *Query showing the use of the* ***IN*** *operator*

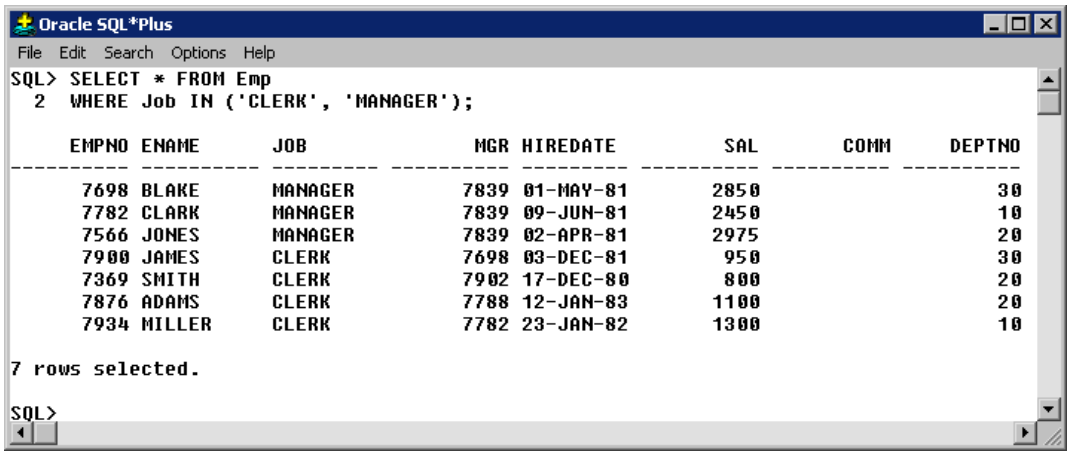


Figure 3-14 *Query showing the use of the* ***IN*** *operator with string inlist*

NOT IN

You can combine the **IN** operator with the **NOT** operator. The **NOT IN** operator works just opposite to the **IN** operator. The syntax for using the **NOT IN** operator is as follows:

```
SELECT Column1, Column2.....
FROM Table
WHERE Column_Name NOT IN (Value1, Value2, Value3,.....);
```

For example:

```
SELECT Ename, Sal, Deptno FROM Emp
WHERE Empno NOT IN (7698, 7782, 7566, 7900, 7934, 7788);
```

The above query will return the **Ename**, **Sal**, and **Deptno** from the **Emp** table of those employees whose Employee number is not 7698, 7782, 7566, 7900, 7934, and 7788. The output of the above query is as follows:

```
ENAME      SAL        DEPTNO
---------- ---------- ----------
KING       5000       10
MARTIN     1250       30
ALLEN      1600       30
TURNER     1500       30
WARD       1250       30
FORD       3000       20
SMITH      800        20
ADAMS      1100       20
```

EXISTS and NOT EXISTS Operators

The **EXISTS** operator is used to check the existence of those rows whose values match with the subquery. The subquery can be a query on the same or different tables, or a combination of both tables used in main query. When a subquery returns a single value, it means that the operator has achieved the target. The syntax for using the **EXISTS** operator is as follows:

```
SELECT Column_Name
FROM Table1
WHERE EXISTS (SELECT Column_Name FROM Table2);
```

The **EXISTS** operator can be used with any valid SQL statement: **SELECT**, **INSERT**, **UPDATE**, or **DELETE**. In most cases, this type of query is used with a standard join to improve performance. The **EXISTS** operator typically provides better performance than the **IN** operator.



Note
You will learn about subqueries later in this chapter.

For example:

```
SELECT Empno, Ename FROM Emp E
WHERE EXISTS (SELECT 1 FROM Dept D
              WHERE E.Deptno = D.Deptno);
```

The output of the above query is as follows:

```
EMPNO      ENAME
---------- ----------
7934       MILLER
7839       KING
7782       CLARK
```

```
7902        FORD
7876        ADAMS
7788        SCOTT
7566        JONES
7369        SMITH
7900        JAMES
7844        TURNER
7698        BLAKE
7654        MARTIN
7521        WARD
7499        ALLEN
```

The following example will illustrate the use of the **EXISTS** operator with the **DELETE** statement:

```
DELETE FROM Emp
WHERE EXISTS (SELECT * FROM Dept
                Where Emp.Deptno = Dept.Deptno);
```

The above query will delete all those records from the **Emp** table, in which the value of **Deptno** of the **Emp** table is same as the value of **Deptno** of the **Dept** table.

The following example will illustrate the use of the **EXISTS** operator with the **UPDATE** statement:

```
UPDATE Emp
SET Deptno= ( SELECT Dept.Deptno FROM Dept
                WHERE Dept.Dname = 'RESEARCH')
WHERE EXISTS ( SELECT Dept.Deptno FROM Dept
                WHERE Dept.Dname = 'RESEARCH');
```

NOT EXISTS

You can also combine the **EXISTS** operator with the **NOT** statement. The **NOT EXISTS** operator works just opposite to the **EXISTS** operator. The syntax for using the **NOT EXISTS** operator is as follows:

```
SELECT Column_Name
FROM Table1
WHERE NOT EXISTS (SELECT Column_Name FROM Table2);
```

For example:

```
SELECT Deptno, Dname FROM Dept D
WHERE NOT EXISTS (SELECT 1 FROM Emp E
                    WHERE D.Deptno = E.Deptno)
```

The above query will return the number and name of the departments from the **Dept** table, in which there are no records of **Deptno** in the **Emp** table. The output of the above query is as follows:

```
DEPTNO          DNAME
--------------  --------------------
40              OPERATIONS
```

Set Operators

Sometimes, you may need to combine the results of two or more **SELECT** statements. Oracle database provides the set operators to meet this requirement.

The set operators are used to combine the data of similar type from more than one query. Oracle SQL supports the following four set operators:

1. UNION ALL
2. UNION
3. MINUS
4. INTERSECT

The SQL statements containing these operators are referred as compound queries and each **SELECT** statement in a compound query is referred to as a composite query. You can combine two **SELECT** statements into a compound query by a set operator. This is possible only when the **SELECT** statement satisfies the following two conditions:

1. The result sets of both the queries must have same number of columns.

2. The data type of each column in the second result set must match the data type of its corresponding column in the first result set.

These conditions are also referred to as union compatibility conditions. The term union compatibility is used here even though these conditions apply to other set operations as well. The set operations are often called as vertical joins because the result is formed by combining the data from two or more **SELECT** statements based on columns, instead of rows. The syntax of a query involving a set operator is as follows:

```
<component query>
{UNION | UNION ALL | MINUS | INTERSECT}
<component query>
```

The keywords **UNION**, **UNION ALL**, **MINUS**, and **INTERSECT** are set operators. You can have more than two component queries in a composite query, but the set operators used in the composite query will always be one less than the number of components used.

The following sections discuss syntax, examples, rules, and restrictions for the four set operators.

UNION ALL Operator

This operator combines the results of two or more queries into a single result set. This operation returns the rows that are retrieved by either of the queries. The **UNION ALL** operator allows the duplicate rows in the result set.

The **UNION ALL** operator is used when you want duplicate rows to occur in the result set. The syntax for using **UNION ALL** is as follows:

```
SELECT statement
UNION ALL
SELECT statement;
```

In the above syntax, the **UNION ALL** operator will join the result set of the two **SELECT** statements.

For example:

```
SELECT Ename FROM Emp
UNION ALL
SELECT Dname FROM Dept;
```

The above example generates a list of names of employee and department from the **Emp** and **Dept** tables.

The following example will illustrate the use of the **UNION ALL** operator with the **ORDER BY** clause:

```
SELECT Empno, Ename FROM Emp
WHERE Empno > 7300
UNION ALL
SELECT Deptno, Dname FROM Dept
WHERE Deptno > 20
ORDER BY 2;
```

Since the column names are different in the two **SELECT** statements, it is more appropriate to reference the columns in the **ORDER BY** clause by their position in the result set. In the above query, the result has been sorted by **Ename** / **Dname** in ascending order, as denoted by the '**ORDER BY 2**'.

Here **Ename** / **Dname** fields are in position 2 in the result set.

UNION Operator

This operator combines the results of two or more queries into a single result set. The single result set consists of distinct rows returned by all queries. The **UNION** operator returns the distinct rows retrieved by either of the queries.

Unlike the **UNION ALL** operator, the **UNION** operator eliminates duplicate rows from the result set. The syntax for using the **UNION** operator is as follows:

```
SELECT statement
UNION
SELECT statement;
```

In the above syntax, the **UNION** operator joins the result sets of two **SELECT** statements and eliminates duplicate rows.

For example:

```
SELECT Dname FROM Dept
UNION
SELECT Ename FROM Emp;
```

The above example will generate a list of distinct names of departments from the **Dept** table and distinct names of employees from the **Emp** table. The **UNION** operator returns only the distinct rows from either of the queries.

The following example will illustrate the use of the **UNION** operator with the **ORDER BY** clause:

```
SELECT Empno, Ename FROM Emp
WHERE Empno > 7300
UNION
SELECT Deptno, Dname FROM Dept
WHERE Deptno > 20
ORDER BY 2;
```

MINUS Operator

The **MINUS** operator is used to return the difference between two sets. This operator returns only those rows that exist in the first query but not in the second query. The syntax for using the **MINUS** operator is as follows:

```
SELECT statement
MINUS
SELECT statement;
```

In the above syntax, the **MINUS** operator joins the result set of the two **SELECT** statements and returns only the rows that are not in the second **SELECT** statement.

For example:

```
SELECT Deptno FROM Dept
MINUS
SELECT Deptno FROM Emp;
```

The above example will generate a list of department numbers from the **Dept** table which are not in the **Emp** table.

The following example will illustrate the use of the **MINUS** operator with the **ORDER BY** clause:

```
SELECT Deptno FROM Dept
MINUS
SELECT Deptno FROM Emp
ORDER BY 1;
```

INTERSECT Operator

The **INTERSECT** operator is used to return all distinct rows returned by the different **SELECT** queries. The syntax for using the **INTERSECT** operator is as follows:

```
SELECT statement
INTERSECT
SELECT statement;
```

In the above syntax, the **INTERSECT** operator joins the result set of the two **SELECT** statements and then returns the distinct result set retrieved by both **SELECT** statements.

For example:

```
SELECT  Deptno FROM Dept
INTERSECT
SELECT  Deptno FROM Emp;
```

The above example will generate a list of distinct department numbers from the **Dept** and **Emp** tables. The **INTERSECT** operator returns only the distinct rows from either of the queries.

The following example will illustrate the use of the **INTERSECT** operator with the **ORDER BY** clause:

```
SELECT  Deptno FROM Dept
INTERSECT
SELECT  Deptno FROM Emp
ORDER BY 1;
```

Rules and Restrictions on Set Operations

The following list summarizes some simple rules, restrictions, and notes on Set operations:

1. Set operators are not applied on the columns of the data type **BLOB**, **CLOB**, **BFILE**, and **VARRAY**. However, they can be applied on the nested table columns.

2. The **UNION**, **INTERSECT**, and **MINUS** operators are not valid on the columns having the data type **LONG**.

3. Set operators are not used with those **SELECT** statements that contain the expression of the **TABLE** collection.

4. The **FOR UPDATE** clause cannot be used with the set operators.

5. The number and size of the columns in the **SELECT** list of the component queries are limited by the block size of the database. The total bytes of the selected columns cannot exceed one database block.

Operator Precedence

Operator precedence refers to the order in which Oracle evaluates different operators within the same expression. If an expression containing multiple operators, Oracle will evaluate the higher precedence operators first before evaluating the lower precedence operators. In case of operators having equal precedence, Oracle evaluates them from left to right within an expression.

Table 3-2 lists the levels of operator precedence from high to low. Operators listed on the same line have the same precedence.

Operator	Operation
+, -	identity, negation (Unary operator)
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, !=, <, >, <=, >=, LIKE, BETWEEN, IN	comparison
NOT	negation
AND	logical AND operation
OR	logical OR operation

Table 3-2 *The SQL operator precedence*

For example:

Consider the following expression:

```
1+2*3
```

In the above expression, Oracle will first multiply 2 by 3 and then add the result to 1 because multiplication has higher precedence than addition.

You can use the parentheses in the above expression to override operator precedence, as given below:

```
(1+2)*3
```

In this expression, Oracle will evaluate the expression inside the parentheses first, then evaluate the expressions outside the parentheses.

ORDER BY Clause

The **ORDER BY** clause allows you to arrange the data retrieved from a table in a sorted order. The rows retrieved are sorted either in the ascending or in the descending order.

The syntax for using the **ORDER BY** clause is as follows:

```
SELECT Column_name FROM Table_name
WHERE Condition
ORDER BY columns ASC/DESC;
```

In the above syntax, **ORDER BY** is the keyword and **Column_name** is the name of column of the table **Table_name**. The result will be sorted depending upon the column or columns specified in the **ORDER BY** clause. The keyword **ASC** indicates that the result set will be sorted in the ascending order and **DESC** indicates that the result set will be sorted in the descending order. If the **ASC** or **DESC** value is omitted, Oracle will assume the ascending order as the default value.

For example:

```
SELECT Ename
FROM Emp
WHERE Job ='MANAGER'
ORDER BY Ename;
```

In the above example, the query will return the names of the employees whose **Job** is **MANAGER**. As discussed earlier, if you omit the keyword **ASC/DESC**, Oracle will take the default value as **ASC** and, therefore, the records will be sorted by the field **Ename** in the ascending order, as shown in Figure 3-15.

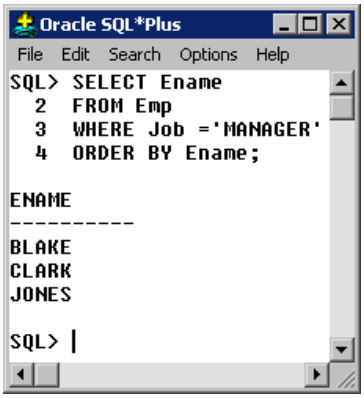


Figure 3-15 *Sorting records by* ***Ename***

```
SELECT Ename FROM Emp
WHERE Job ='MANAGER'
ORDER BY Ename DESC;
```

The above query will return the names of employees, whose **Job** is **MANAGER**. Here, the records will be sorted by the field **Ename** in the descending order, as shown in Figure 3-16.

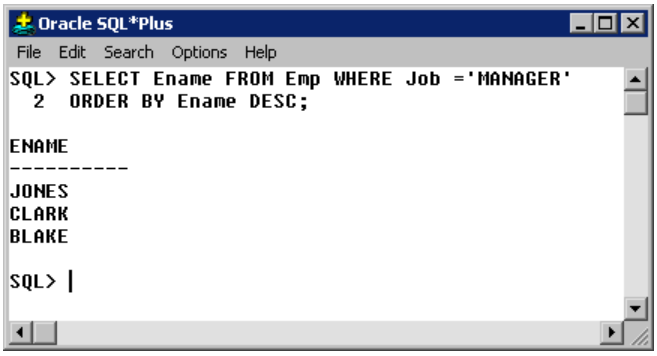


Figure 3-16 *Sorting records by* ***Ename*** *(****DESC****)*

You can also sort records by position of the fields in the result set, where the first field is on position 1 and the next field is on position 2, and so on.

For example, the query

```
SELECT Ename, Sal
FROM Emp
WHERE Job ='MANAGER'
ORDER BY 1 DESC;
```

and

```
SELECT Ename, Sal
FROM Emp
WHERE Job ='MANAGER'
ORDER BY 2 DESC;
```

sort the records by the position fields.

The above queries will return all records sorted by the position of the field in the descending order.

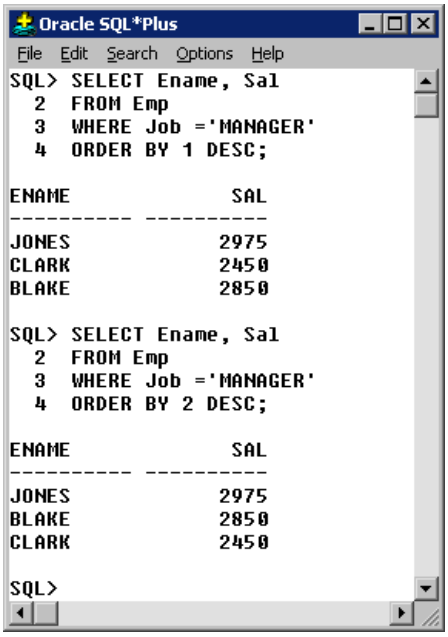


Figure 3-17 *Sorting records by the position field*

The first query will sort the records based on the **Ename** field because **Ename** is on position 1 in the query. In the second query, the records will be sorted by the field **Sal** because **Sal** is on position 2 in the query. The output of the above queries is shown in Figure 3-17.

The following queries will sort the records by specifying two fields in the **ORDER BY** clause:

```
SELECT Ename, Sal
FROM Emp
WHERE Job= 'MANAGER'
ORDER BY Ename DESC, Sal ASC;
```

The result set of the above query will be sorted twice, first by the **Ename** field and second by the **Sal** field, as shown in Figure 3-18.

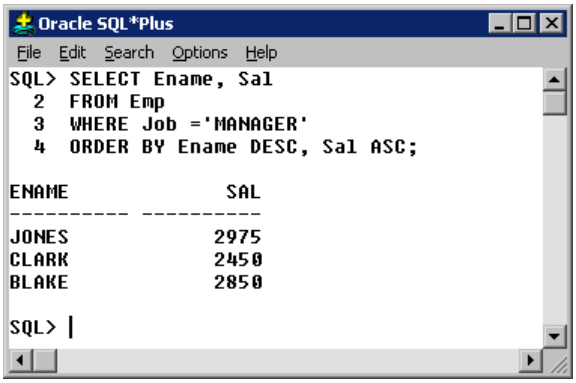


Figure 3-18 *Sorting the records by two fields*

GROUP BY Clause

The **GROUP BY** clause is used in the **SELECT** statement to collect data from multiple records and group the results that have matching values for one or more columns. The syntax for using the **GROUP BY** clause is as follows:

```
SELECT Column1, Column2, ..., Column-n
FROM Table_name
WHERE Condition
GROUP BY Column1, Column2, ..., Column-n;
```

In the above syntax, the **GROUP BY** is a keyword and **Column1**, **Column2**, and **Column-n** are the names of columns of the table **Table_name**. You can group the result set by one or more columns.

You can also use the aggregate function in the **SELECT** statement with the **GROUP BY** clause. The syntax for using the **GROUP BY** clause while using the aggregate function in the **SELECT** statement is as follows:

```
SELECT Column1, Column2, ..., Column-n, aggregate_Function(Expression)
FROM Table_name
WHERE Condition
GROUP BY Column1, Column2, ..., Column-n;
```

In the above syntax, the **aggregate_Function** can be any aggregate function such as **SUM**, **COUNT**, **MIN**, or **MAX**. These aggregate functions will be discussed in the later chapter.

Given below are some examples showing the use of the **GROUP BY** clause with different aggregate functions.

The following example will illustrate the use of the **GROUP BY** clause with the **SUM** function:

```
SELECT Job, SUM(Sal)
FROM Emp
GROUP BY Job;
```

In the above example, the SQL query will return the job description of employees along with the total salary (for example, total salary of the employees having job description CLERK), as shown in Figure 3-19.

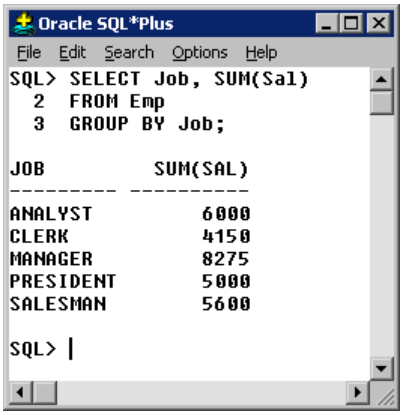


Figure 3-19 *The **GROUP BY** clause with the **SUM** function*

The following example will illustrate the use of the **GROUP BY** clause with the **COUNT** function.

```
SELECT Job, COUNT(*)
FROM Emp
GROUP BY Job;
```

The above SQL query will return the job description of employees with the total number of employees in each job description. Figure 3-20 shows the output of this query.

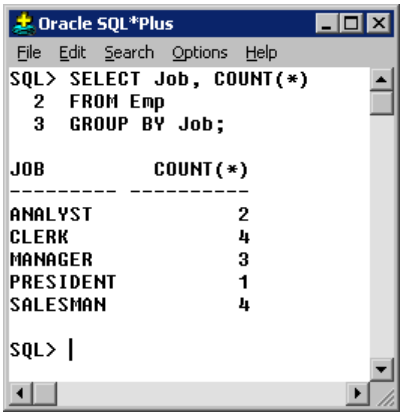


Figure 3-20 *The **GROUP BY** clause with the **COUNT** function*

The following example will illustrate the use of the **GROUP BY** clause with the **MIN** function:

```
SELECT  Job, MIN(Sal)
FROM Emp
GROUP BY Job;
```

In the above example, the SQL query will return the job description of employees with the minimum salary in each job description from the **Emp** table.

The following example will illustrate the use of the **GROUP BY** clause with the **MAX** function:

```
SELECT  Job, MAX(Sal)
FROM Emp
GROUP BY Job;
```

In the above example, the SQL query will return the job description of the employees with the maximum salary in each job description from the **Emp** table.

HAVING Clause

The **HAVING** clause is used in the **SELECT** statement to filter the data returned by the **GROUP BY** clause. The **HAVING** clause is similar to the **WHERE** clause and it is evaluated once Oracle has evaluated the grouped values. The syntax for using the **HAVING** clause is as follows:

```
SELECT Column1, Column2, ....., Column-n
FROM Tables
WHERE Condition
GROUP BY Column1, Column2, ......, Column_n
HAVING SearchCondition;
```

In the above syntax, the **SearchCondition** is a boolean expression, and it can contain only grouping columns that means the columns that are part of the aggregate expression and the columns that are part of a subquery. Consider the following query:

```
SELECT Job, Sal FROM Emp
GROUP BY Job
HAVING Empno >= 7521;
```

The above query is not valid because **Empno** is not a grouping column, it is not a part of the aggregate expression, and it does not appear in the subquery. Therefore, this query will return an error with the message **not a GROUP BY expression**, as shown in Figure 3-21.

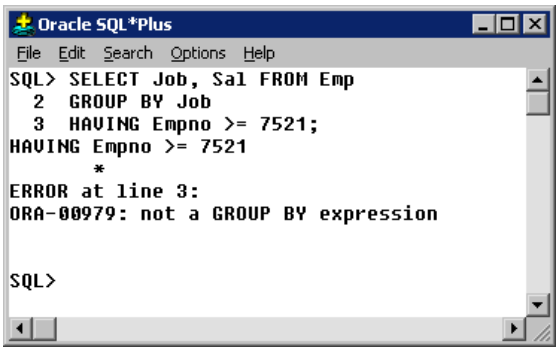


Figure 3-21 *Invalid use of the* ***HAVING*** *clause*

You can also use the aggregate function in the **SELECT** statement with the **HAVING** clause. The syntax for using the **HAVING** clause with an aggregate function in the **SELECT** statement is as follows:

```
SELECT column_name, aggregate_function(expression/column_name)
FROM Table_name
WHERE SearchCondition
GROUP BY column_name
HAVING SearchCondition;
```

In the above syntax, the **aggregate_function** can be any aggregate function such as **SUM**, **COUNT**, **MIN**, **MAX**, and so on.

Some examples showing use of the **HAVING** clause with the different aggregate functions are as follows:

The following example will illustrate the use of the **HAVING** clause with the **AVG** function:

```
SELECT Job, AVG(Sal)
FROM Emp
GROUP BY Job
HAVING AVG(Sal)>2000;
```

In the above SQL query, the **AVG** function will return the average salary of the employees. The **HAVING** clause will filter the results returned by the **GROUP BY** clause. As a result, this query will return those job descriptions salary whose average salary is greater than 2000, as shown in Figure 3-22.

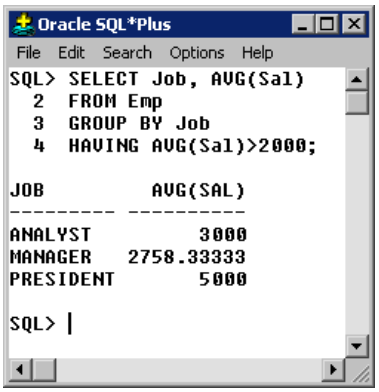


Figure 3-22 *Using the **AVG** function*

The following example will illustrate the use of the **HAVING** clause with the **MAX** function:

```
SELECT Deptno, MAX(Sal)
FROM Emp
GROUP BY Deptno
HAVING MAX(Sal) > 4000;
```

In the above SQL query, the **MAX** function will return the maximum salary of the employees. The **HAVING** clause will filter the results returned by the **GROUP BY** clause. As a result, this query will return those department numbers in which maximum salary of an employee is greater than 4000, as shown in Figure 3-23.

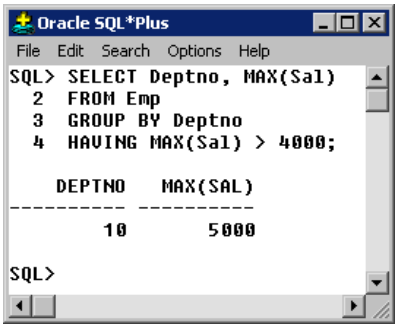


Figure 3-23 *Using the HAVING clause with the **MAX** function*

The following example will illustrate the use of the **HAVING** clause with the **MIN** function:

```
SELECT Deptno, MIN(Sal)
FROM Emp
GROUP BY Deptno
HAVING MIN(Sal) > 1000;
```

In the above SQL query, the **MIN** function will return the minimum salary of the employees. The **HAVING** clause will filter the results returned by the **GROUP BY** clause. As a result, this query will return those department numbers in which minimum salary of an employee is greater than 1000.

The following example will illustrate the use of the **HAVING** clause with the **SUM** function:

```
SELECT Deptno, SUM(Sal) FROM Emp
GROUP BY Deptno
HAVING SUM(Sal) > 5000;
```

In the above query, the **SUM** function will return the total salary of each department. The **HAVING** clause will filter the result set and return the department numbers having total salary greater than 5000.

IS NULL and IS NOT NULL Operators

The **IS NULL** and **IS NOT NULL** operators are used to find the NULL and not NULL values respectively. The **IS NULL** operator returns TRUE, when the value is NULL; and FALSE, when the value is not NULL. The **IS NOT NULL** operator returns TRUE, when the value is not NULL; and FALSE, when the value is NULL.

The following example will illustrate the use of the **IS NULL** operator:

```
SELECT * FROM Emp
WHERE Comm IS NULL;
```

The above SQL query will return all records from the **Emp** table where **Comm** contains a NULL value.

The following example will illustrate the use of the **IS NOT NULL** operator**:**

```
SELECT * FROM Emp
WHERE Comm IS NOT NULL;
```

The above SQL query will return all records from the **Emp** table where **Comm** does not contain a NULL value.

SELECTING DATA FROM THE DUAL TABLE

The **Dual** table is the default table in the Oracle database. It is created by Oracle along with the data dictionary. It is a special one-row and one-column table. The **Dual** table has exactly one column called **DUMMY** of VARCHAR2(1) data type which has a single row with a value of X (here, X can be any value), as shown in Figure 3-24. The owner of the dual table is **SYS** but it can be accessed by every user in the Oracle database.

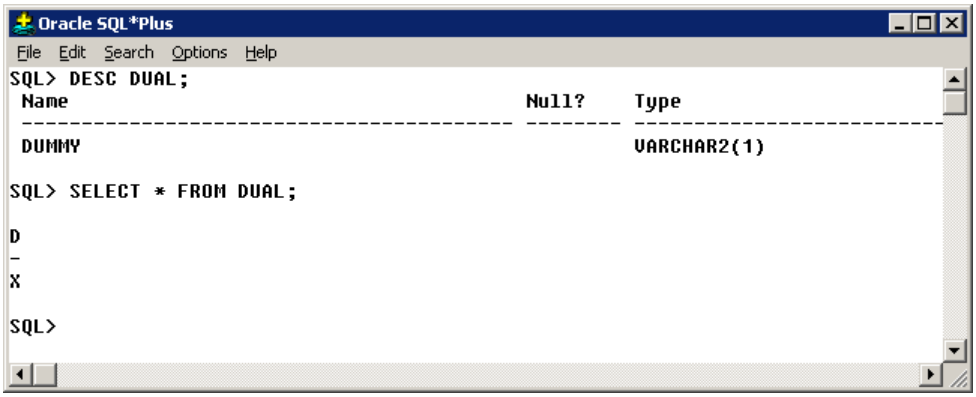


*Figure 3-24 The **DUAL** table*

The following example will illustrate the use of the **DUAL** table to find the system date and the current user:

```
SELECT SYSDATE
FROM DUAL;

SELECT USER
FROM DUAL;
```

In the above example, the **SELECT** statement query will return the system date and the second **SELECT** statement will return the name of current user, as shown in Figure 3-25. The **SYSDATE** function is used to return the system date and is discussed in the later chapters.

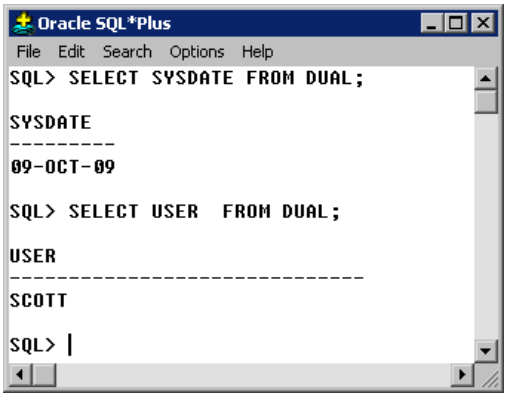


Figure 3-25 *Selecting SYSDATE and USER from the* ***DUAL*** *table*

SUBQUERIES

Query within a query is called a subquery. The statement containing a subquery is called the parent statement. Subqueries are used to retrieve data from tables and the retrieved data depends on the value in the table itself. The output of a subquery is the input to the main query and on the basis of output of the subquery, the result set of whole query is generated.

The syntax for using a subquery is as follows:

```
SELECT Column1, Column2,.....
FROM Table_name
WHERE Column_X operator   (SELECT Column_Names
                           FROM Table_name
                           WHERE Search_Condition);
```

In the above syntax, the **SELECT** statement appearing within parenthesis is a subquery, and the rest of the query is the main query. The output of the subquery is the input to the main query. The **WHERE** clause that appears in the subquery is optional. Here, subquery can return single or multiple values. Subqueries can be used in the **INSERT**, **UPDATE** and **DELETE** statements.

For example:

```
SELECT Empno, Ename, Sal, Deptno FROM Emp
WHERE Sal > (SELECT Sal FROM Emp
             WHERE Ename = 'JONES');
```

In the above query, the inner query will return the salary of the employee named **JONES**. This salary will be compared with the outer query and then only those rows will be returned that meet the condition in the **WHERE** clause. The output of the above query is as follows:

```
EMPNO      ENAME      SAL        DEPTNO
---------- ---------- ---------- ----------
7839       KING       5000       10
7902       FORD       3000       20
7788       SCOTT      3000       20
```

Subqueries can be of three types: single-row, multiple-row, and multiple-column subqueries. These types of subqueries are discussed next.

Single-Row Subqueries

Single-row subqueries return only one row as a result. The operators that can be used with single-row subqueries are =, >, >=, <, <=, and <>.

Given below is a list of examples that illustrates the use of single-row subqueries in different conditions.

In order to list the employees who earn less than the average salary in any organization, the group function **AVG** must be used to calculate the average salary of employees. However, the group function cannot be used with the **WHERE** clause. In such a case, you can use a subquery.

```
SELECT Empno, Ename, Sal, Hiredate FROM Emp
WHERE Sal > (SELECT AVG(Sal) FROM Emp);
```

In the above example, the main query will return the details of all those employees whose salary is greater than the average salary. If subquery returns more than one value, the **IN** operator must be used. The output of the above query is as follows:

```
EMPNO      ENAME      SAL        HIREDATE
---------- ---------- ---------- ----------
7839       KING       5000       17-NOV-81
7698       BLAKE      2850       01-MAY-81
7782       CLARK      2450       09-JUN-81
7566       JONES      2975       02-APR-81
7902       FORD       3000       03-DEC-81
7788       SCOTT      3000       09-DEC-82
```

The following example will illustrate the use of the **AVG** function with a subquery:

```
SELECT Emp.*, Dept.Dname FROM Emp, Dept
WHERE Emp.Deptno = Dept.Deptno AND
Emp.Sal > (SELECT AVG(Sal) FROM Emp);
```

In the above example, the main query will return all fields from the **Emp** table and only one field called **Dname** from the **Dept** table. The query will return only those records in which the salary of an employee is greater than the average salary of employees and has matching values for the column **Deptno** in both **Emp** and **Dept** tables.

The following query will illustrate the use of the join operation in a subquery:

```
SELECT Emp.Ename, Dept.Dname FROM Emp INNER JOIN Dept
ON Emp.Deptno = Dept.Deptno
WHERE Emp.Sal > (SELECT AVG(Sal) FROM Emp);
```

In the above example, the main query will return the names of employees from the **Emp** table as well as department names of employees from the **Dept** table.

Multiple-Row Subqueries

The subquery that returns multiple rows is called a multiple-row subquery. You need to use the comparison operators IN, ALL, and ANY to handle the multiple rows returned by the subquery.

The following example will illustrate the use of the **IN** operator with a subquery:

```
SELECT * FROM Emp
WHERE Empno IN (SELECT Empno FROM Emp
                WHERE Comm >= 150);
```

In the above example, the main query will return more than one record because the inner query will return more than one value. Also, the subquery will return those employee numbers from the **Emp** table whose **Comm** is greater than or equal to 150.

Multiple-Column Subqueries

A multiple-column subquery returns more than one column. In a multiple-column subquery, the resulting rows of the subquery are evaluated pair-wise (that is column to column and row to row comparisons) in the main query.

For example:

```
SELECT Empno, Ename, Deptno FROM Emp
WHERE (Empno, Deptno) IN(SELECT e.Empno, D.Deptno
                         FROM Emp E, Dept D);
```

In the above query, the inner query returns two columns, Empno and Deptno. Here, the comparison is column to column that means the column values are compared as a pair and not individually. The output of the above query is as follows:

```
EMPNO       ENAME       DEPTNO
----------  ----------  --------------
7839        KING        10
7698        BLAKE       30
7782        CLARK       10
7566        JONES       20
7654        MARTIN      30
7499        ALLEN       30
.....
.....
```

Using a Subquery with the CREATE and INSERT Statements

Subqueries can also be used with the **CREATE** and **INSERT** Statement.

For example:

```
CREATE TABLE Emp_Dup AS SELECT * FROM Emp;
```

The above query will create the table **Emp_Dup** from the existing table **Emp** and copy all records from the **Emp** table.

Moreover, you can create a table without copying the rows from an existing table. For example:

```
CREATE TABLE Emp_Dup AS SELECT * FROM Emp WHERE 1>2;
```

Now, insert the records into the **Emp_Dup** table using the **INSERT** statement:

```
INSERT INTO Emp_Dup(Empno, Sal, Deptno)
(SELECT Empno, Sal, Deptno FROM Emp
WHERE Deptno = 10);
```

The above query will insert values into three columns of the **Emp_Dup** table from the **Emp** table where **Deptno** is **10**.

Using a Subquery to Delete and Update Data

Subqueries are not only used to retrieve or insert data into tables but also to delete or update the data in the tables.

For example:

```
DELETE FROM Emp WHERE Empno IN
(SELECT Empno FROM Emp WHERE Comm <150);
```

In the above example, the inner query will return only those employee numbers whose commission is less than 150. The **DELETE** statement will delete only those records from the **Emp** table where **Empno** exists in the employee numbers returned by the inner query.

The following example will illustrate the use of a subquery to update the existing records:

```
UPDATE Emp_Dup SET (Sal, Comm)=(SELECT AVG(Sal), SUM(Comm)
FROM Emp) WHERE Empno= 7934;
```

In the above example, the query updates the salary and commission of an employee whose employee number is **7934** in the **Emp_Dup** table.

CORRELATED SUBQUERIES

A correlated subquery is the SELECT statement that is nested inside another query containing the reference of one or more columns in the outer query.

For example:

```
SELECT Empno, MGR, Ename, Sal,
FROM Emp OuterE
WHERE Sal > (SELECT AVG(Sal) FROM Emp InnerE
             WHERE InnerE.Empno = Empno);
```

In the above correlated subquery, you can see that inner query contains a reference to **InnerE.Empno**. This reference compares the outer query's **Empno** with the inner query's **Empno**. When the above query is executed, the Oracle will execute the inner query for each employee record. The inner query will calculate the average salary of the particular employee for the row being processed in the outer query. This correlated subquery determines, if the inner query returns a value that meets the condition of the **WHERE** clause. The output of the above query is as follows:

EMPNO	MGR	ENAME	SAL
7839		KING	5000
7698	7839	BLAKE	2850
7782	7839	CLARK	2450
7566	7839	JONES	2975
7902	7566	FORD	3000
7788	7566	SCOTT	3000

JOIN

Sometimes you may need to retrieve records from more than one table. To do so, the Oracle database provides a technique called Join. Joins are used to combine the result set of one or more tables. A join operation can be performed whenever two or more tables are listed in the **FROM** clause of an SQL statement. In order to query data from more than one table, you

need to identify common columns that relate the tables. If any two of these tables have a common column name, then you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join means accessing rows from one or more tables. A join operation is essential while retrieving data from one or more tables. The general syntax of a **SELECT** query that joins two tables is as follows:

```
SELECT Column1, Column2,.....
FROM Table1, Table2
WHERE Table1.Join_Column = Table2.Join_Column;
```

In the above syntax, the **SELECT** clause lists the columns that you want to retrieve and the **FROM** clause lists all table names that are involved in the join operation. On the basis of the join condition (**Table1.Join_Column = Table2.Join_Column**), the rows from the tables **Table1** and **Table2** will be retrieved. This means that only those rows will be retrieved from the tables that meet the join condition. If you want to retrieve a column that exists in more than one table, you need to qualify the column name in the **SELECT** clause, so that Oracle returns the specific column. To qualify a column in the **SELECT** clause, you have to specify the table name containing the column, followed by a period (.) and column name. Joins are of various types and these are discussed next.

INNER JOIN

INNER JOIN joins two or more tables and returns only those rows from the tables that follow the join condition. The syntax for using the **INNER JOIN** is as follows:

```
SELECT Column1, Column2,...
FROM Table1, Table2,...
WHERE Table1.Column1=Table2.Column1...
```

In the above syntax, the join condition (**Table1.Column1=Table2.Column1**) appears in the **WHERE** clause.

For example:

```
SELECT Empno, Ename, Dname
FROM Emp, Dept
WHERE Emp.Deptno = Dept.Deptno;
```

The above SQL query will return the number, name and department name of employees. It will return only those rows where the department number of the table **Emp** matches with department number of the table **Dept**.



Note
*You can add more than one condition in the **WHERE** clause.*

The syntax for using the ISO/ANSI **INNER JOIN** is as follows:

```
SELECT Column1, Column2,...
FROM Table1 [INNER JOIN][JOIN] Table2
[ON][USING] Table1.Column1=Table2.Column1
```

In the above syntax, the **SELECT** clause lists columns and the **FROM** clause lists the tables involved in the join operation.

JOIN and INNER JOIN
These keywords indicate that the join operation is being performed. This clause is used to replace the comma-delimited used between tables in the **FROM** clause.

ON condition
The **ON** clause is used to specify a join condition. This clause is used to replace the join condition in the **WHERE** clause.

For example:

```
SELECT Empno, Ename, Dname
FROM Emp INNER JOIN Dept
ON Emp.Deptno=Dept.Deptno;
```

The above SQL query will return only those rows, where the department number in the **Emp** table matches with the department number in the **Dept** table.

USING (column)
This clause is also used to replace the join condition in the **WHERE** clause. This clause is used when several columns share the same name in tables that appear in the **FROM** clause. It is recommended not to qualify the column name with a table name or table alias within this clause.

For example:

```
SELECT Ename, Sal, Dname
FROM Emp INNER JOIN Dept
USING(Deptno);
```

The above SQL query will return only those rows from the tables **Emp** and **Dept**, where the department numbers match.

The following example will the illustrate use of **INNER JOIN** with the **WHERE** clause:

```
SELECT Empno, Ename, Dname
FROM Emp INNER JOIN Dept
ON Emp.Deptno=Dept.Deptno
WHERE Emp.Job IN('MANAGER', 'SALESMAN', 'ANALYST');
```

The above SQL query will return only those rows where the department number of the **Emp** table matches with the department number of the **Dept** table provided the employee's job is **MANAGER**, **SALESMAN**, or **ANALYST**. You can also use the **WHERE** clause in the ISO/ANSII **INNER JOIN** semantics for further filtering of records.

OUTER JOIN

The **OUTER JOIN** returns all rows of a table with only those rows from another table that follow the join condition. It also returns a null value in place of the records which do not follow the join condition from the another table. There are three types of outer joins: **LEFT OUTER JOIN**, **RIGHT OUTER JOIN**, and **FULL OUTER JOIN**. These types are discussed next.

LEFT OUTER JOIN

The **LEFT OUTER JOIN** returns all rows of the first table (the table that appears first in the table list of the **FROM** clause) and only those rows from the second table that follow the join condition. It also returns replacement of the non-matching (that does not follow the join condition) rows from the second table with a NULL value. The syntax for using the **LEFT OUTER JOIN** is as follows:

```
SELECT Column1, Column2,...
FROM Table1 LEFT OUTER JOIN Table2
ON Table1.Join_Column=Table2.Join_Column
```

In the above syntax, the **SELECT** clause lists the columns to be displayed and the **FROM** clause lists the tables involved in the join operation with the table aliases. The **LEFT OUTER JOIN** is a keyword that indicates that the left outer join operation is being performed. This syntax is used to replace the comma-delimiter used between tables in the **FROM** clause. The **ON** clause in the syntax is used to specify a join condition. This syntax is used to replace the join condition in the **WHERE** clause.

For example:

```
SELECT Emp.Ename, Dept.Dname
FROM Emp LEFT OUTER JOIN Dept
ON Emp.Deptno=Dept.Deptno ;
```

The above SQL query will return all rows from the **Dept** table and only those rows from the **Emp** table that meet the join condition. Also, it will return the NULL value for those rows that do not follow the join condition.

RIGHT OUTER JOIN

The **RIGHT OUTER JOIN** returns all rows of the second table (that appears second in the table list of the **FROM** clause) and only those rows from the first table that follow the join condition. It also returns the replacement of the non-matching rows (rows that do not follow

the join condition) from the first table with a NULL value. The syntax for using the **RIGHT OUTER JOIN** is as follows:

```
SELECT Column1, Column2,...
FROM Table1 RIGHT OUTER JOIN Table2
ON Table1.Join_Column=Table2.Join_Column
```

In the above syntax, the **SELECT** clause lists the columns to be displayed and the **FROM** clause lists the tables involved in the join operation.

The **RIGHT OUTER JOIN** is a keyword and indicates that the join operation is being performed. This syntax is used to replace the comma-delimited table expressions used in the **FROM** and **WHERE** clauses.

The **ON** clause specifies a join condition. This syntax is used to replace the join condition in the **WHERE** clause.

For example:

```
SELECT Ename, Dname
FROM Emp RIGHT OUTER JOIN Dept
ON Emp.Deptno=Dept.Deptno;
```

The above SQL query will return all rows from the **Emp** table and only those rows from the **Dept** table that meet the join condition. Also, it will return the NULL value for those rows that do not follow the join condition.

FULL OUTER JOIN

The **FULL OUTER JOIN** returns all those rows from both the tables, where the rows from one table match with the rows from the other table. The syntax for using the **FULL OUTER JOIN** is as follows:

```
SELECT Column1, Column2,...
FROM Table1 FULL OUTER JOIN Table2
ON Table1.Join_Column=Table2.Join_Column
```

In the above syntax, the **SELECT** clause lists the columns to be displayed and the **FROM** clause lists the tables involved in the join operation.

For example:

```
SELECT Emp.Ename, Emp.Sal, Dept.Dname
FROM Emp FULL OUTER JOIN Dept
ON Emp.Deptno=Dept.Deptno ;
```

The above query will display two columns **Ename** and **Sal** from the table **Emp**, and **Dname** from the table **Dept**. It will return only those rows in which the values of the column **Deptno**

of the **Emp** table matches with the values of the column **Deptno** of the **Dept** table. It will also return Null values from both the tables those does not match the join condition.

Self Join

The self join joins a table to itself. It means that a self join joins one row of a table with another row in the same table. It compares one row of a table to itself or with the other rows in the same table. This table appears twice or more times in the **FROM** clause and is followed by table aliases that qualify column names in the join condition and the **SELECT** clause. The syntax for using the self join is as follows:

```
SELECT Column1, Column2,...
FROM Table1 Table_alias1, Table1 Table_alias2, ...
WHERE Table_alias1.Column1=Table_alias2.Column1...
```

In the above syntax, the join condition appears in the **WHERE** clause. The **Table_alias1** and **Table_alias2** refer to the name of **Table1**. Also, **Table_alias1.Column1** and **Table_alias2.Column1** refer to **Column1** from **Table1**.



Note

*For joining a table with itself, you must use an alias for each of the tables in the **FROM** clause as well as in the **SELECT** list and the **WHERE** clause.*

For example:

```
SELECT m.Ename || ' Is manager of ' || e.Ename FROM Emp m, Emp e
WHERE m.MGR = e.Empno;
```

The above query will return both employee number and employee name from the **Emp** table, as here the selfjoin retrieves rows from the same table. The output of the above query is as follows:

```
M.ENAME||'ISMANAGEROF'||E.ENAME
---------------------------------------------------------------
SMITH Is manager of SMITH
ALLEN Is manager of ALLEN
WARD Is manager of WARD
JONES Is manager of JONES
MARTIN Is manager of MARTIN
BLAKE Is manager of BLAKE
CLARK Is manager of CLARK
SCOTT Is manager of SCOTT
KING Is manager of KING
TURNER Is manager of TURNER
ADAMS Is manager of ADAMS
JAMES Is manager of JAMES
FORD Is manager of FORD
MILLER Is manager of MILLER
```

Equijoin

An equijoin contains equality operator (=) in the join condition which is used to match rows from different tables.

For example:

```
SELECT e.Empno, e.Ename, e.Sal, d.Deptno, d.Dname FROM
Emp e, Dept d WHERE e.Deptno = d.Deptno;
```

The above query will return only those rows in which the department number of the **Emp** table matches with the department number of the **Dept** table. The output of the above query is as follows:

EMPNO	ENAME	SAL	DEPTNO	DNAME
7782	CLARK	2450	10	ACCOUNTING
7839	KING	5000	10	ACCOUNTING
7934	MILLER	1300	10	ACCOUNTING
7369	SMITH	800	20	RESEARCH
7876	ADAMS	1100	20	RESEARCH
7902	FORD	3000	20	RESEARCH
7788	SCOTT	3000	20	RESEARCH
7566	JONES	2975	20	RESEARCH
7499	ALLEN	1600	30	SALES
7698	BLAKE	2850	30	SALES
7654	MARTIN	1250	30	SALES
7900	JAMES	950	30	SALES
7844	TURNER	1500	30	SALES
7521	WARD	1250	30	SALES

Cartesian Joins

The cartesian join occurs when you select data from two tables and there is no join condition. It is a join of every row of a table with every row of another table. This only happens, when no matching join columns are specified in the join condition for the table listed in the **FROM** clause. For example, if you have two tables, namely XYZ with 100 rows, and ABC with 200 rows, then the cartesian join will return 20,000 rows.

Consider the following query:

```
SELECT * FROM Emp, Dept;
```

The above query will return each row of the **Emp** table with each row of the **Dept** table. Now consider the following query with the **WHERE** clause:

```
SELECT *  FROM Emp, Dept
WHERE Dept.Deptno = 10
AND Emp.Sal > 2000;
```

The above query will return the details of employees of the department 10 having salary greater than 2000 from the **Emp** and **Dept** tables, as shown in Figure 3-26.

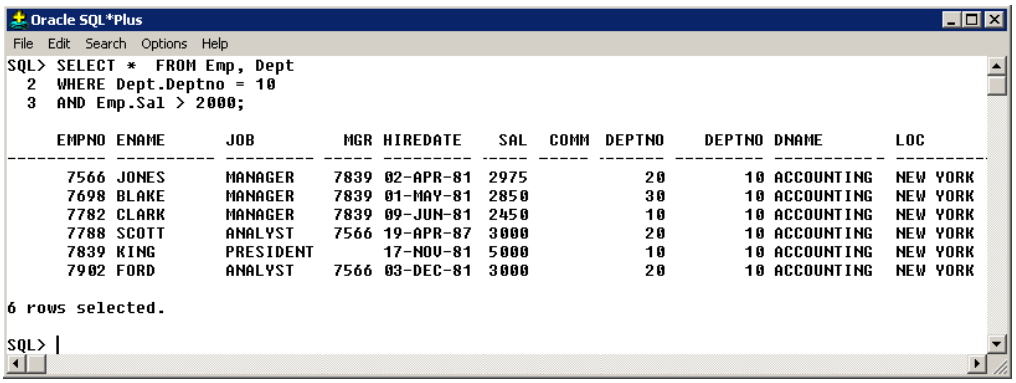


Figure 3-26 *Output of the cartesian join between the* ***Emp*** *and* ***Dept*** *tables*

Antijoins

An antijoin between two tables returns those rows from the first table for which there are no corresponding rows in the second table. It implies that antijoin returns the rows that fail to match the rows returned by the subquery on the right side. Antijoins are written using the **NOT EXISTS** or **NOT IN** operator.

For example:

```
SELECT * FROM Emp
WHERE Deptno NOT IN
        (SELECT Deptno FROM Dept
        WHERE Loc = 'NEW YORK');
```

Semijoins

A semijoin between two tables returns the rows from the first table having one or more matches in the second table. Semijoins are written using the **EXISTS** or **IN** operator.

For example:

```
SELECT * FROM Dept
WHERE EXISTS
        (SELECT * FROM Emp
        WHERE Dept.Deptno = Emp.Deptno)
        ORDER BY Dname;
```

The above query will return the list of departments that have at least one employee. The department name will appear only once in the query output, no matter how many employees it has.

Table Alias Names

Table alias refers to a different name for a table for the purpose of evaluating the query and is most often used in a correlated query. You can code the query with an alias for the table name to make the query easier to code.

For example:

Consider the query given below to retrieve data from two tables:

```
SELECT Emp.Ename, Dept.Dname
FROM Emp RIGHT OUTER JOIN Dept
ON Emp.Deptno=Dept.Deptno ;
```

This query can be coded with the alias for the table name as follows:

```
SELECT E.Ename, D.Dname
FROM Emp E RIGHT OUTER JOIN Dept D
ON E.Deptno=D.Deptno ;
```

In the above example, the table **Emp** is referred by the alias **E** and the table **Dept** is referred by the alias **D**. The above query will return only those rows in which the values of the column **Deptno** of the table **Emp** match with values of the column **Deptno** of the table **Dept**.

Column Alias Names

Column alias refers to the different name for a database column expression and this alias is used for column headings. It does not affect the actual column name. It can be used to show the name of the column according to the user requirement.

For example:

Consider the query given below:

```
SELECT Empno, Ename
FROM Emp;
```

This query can be coded with the alias for the table name in the following way:

```
SELECT Empno “Number”, Ename “Name”
FROM Emp;
```

The above query will display two columns from the table **Emp**. The first column will have the heading **Number** and the other column will have the heading **Name**.

```
SELECT Empno “Number”, Sal “Basic Salary”, Sal + NVL(Comm, 0) “Net Salary”
FROM Emp WHERE Sal >= 3000;
```

The above query will display three columns from the table **Emp**. The first column will have the heading **Number**, the second column will have the heading **Basic Salary**, and the third column will have the heading **Net Salary**. The output of the above query is as follows:

```
Number      Basic Salary    Net Salary
----------  ------------    ----------
7839        5000            5000
7902        3000            3000
7788        3000            3000
```



Note
*The **NVL** function will be discussed in later chapters.*

ACCEPTING VALUES AT RUNTIME

To create an interactive SQL statement, you can define variables in the SQL statement. This allows the users to supply values at runtime, thus enhancing the ability to reuse your scripts. SQL*Plus lets you define variables in your scripts. An ampersand (&), followed by a variable name, prompts for and accepts values at runtime.

For example:

The following **SELECT** statement queries the **Emp** table based on the department number supplied at runtime.

```
SQL>  SELECT Empno, Ename, Job, HireDate, Sal FROM Emp
      WHERE Deptno = &Deptno;
```

The above query will prompt a message **Enter value for deptno** and the value entered by you will be assigned to the **Deptno** variable. The output of the above query is as follows:

```
Enter value for deptno: 30
old   2: WHERE Deptno = &Deptno
new   2: WHERE Deptno = 30

EMPNO       ENAME       JOB         HIREDATE        SAL
----------  ----------  ---------   ----------------  ----------
7499        ALLEN       SALESMAN    20-FEB-81       1600
7521        WARD        SALESMAN    22-FEB-81       1250
7654        MARTIN      SALESMAN    28-SEP-81       1250
7698        BLAKE       MANAGER     01-MAY-81       2850
7844        TURNER      SALESMAN    08-SEP-81       1500
7900        JAMES       CLERK       03-DEC-81       950

6 rows selected.
```

While using substitution variables for the character or date values, you need to enclose the

variables in single quotes. Otherwise, the user will have to enclose them in quotes at runtime. If the variables are not enclosed in single quotes, Oracle considers any non-numeric value as a column name.

Using Substitution Variables

Suppose that you have defined DEPT as a variable in your script, but you want to avoid the prompt for the value at runtime. To do so, you can define a substitution variable in SQL *Plus using the **DEFINE** command. This variable will always have the CHAR data type associated with it.

For example:

```
SQL> DEFINE Deptno = 20;
SQL> DEFINE Deptno;
DEFINE DEPTNO         = "20" (CHAR)

SQL> SELECT * FROM Emp
2  WHERE Deptno = &Deptno;
old   2: WHERE Deptno = &Deptno
new   2: WHERE Deptno = 20
```

The output of the query is as follows:

```
EMPNO ENAME   JOB          MGR    HIREDATE    SAL     COMM   DEPTNO
---------- ----------   ---------  ---------- ----------- ---------- ----------
7369    SMITH    CLERK      7902    17-DEC-80     800                20
7566    JONES    MANAGER    7839    02-APR-81    2975                20
7788    SCOTT    ANALYST    7566    19-APR-87    3000                20
7876    ADAMS    CLERK      7788    23-MAY-87    1100                20
7902    FORD     ANALYST    7566    03-DEC-81    3000                20
```

Saving a Variable for a Session

Consider the following SQL query saved to a file named *Demo_Ex.sql*. When you execute this script file, you will be prompted to enter a value for **COL1**, **COL2**, and **COL3** multiple times:

```
SQL> SELECT &COL1, &COL2, &FROM &TABLE_NAME
2  WHERE &COL4 = &VAL
3
SQL> SAVE Demo_Ex
Created file Demo_Ex.sql
```

In the above example, the SQL statement is saved in the file named *Demo_Ex.sql*. Now, you can execute this file by using the @ or the **START** command as shown below:

```
SQL> START Demo_Ex
Enter value for col1: Empno
```

```
Enter value for col2: Ename
Enter value for col3: Sal
Enter value for table_name: Emp
old   1: SELECT &COL1, &COL2, &COL3 FROM &TABLE_NAME
new   1: SELECT Empno, Ename, Sal FROM Emp
Enter value for col4: Deptno
Enter value for val: 10
old   2: WHERE &COL4 = &VAL
new   2: WHERE Deptno = 10

EMPNO          ENAME          SAL
----------     ----------     ----------
7782           CLARK          5981.45
7839           KING           12207.04
7934           MILLER         3173.83
```

Using Positional Notation for Variables

Instead of variable names, you can also use the positional notation. In this notation, the values are assigned to variables on the basis of their positions and the variables are identified by &1, &2, and so on. You can use this notation by using an ampersand (&), followed by a numeral in the place of a variable name. Consider the following query:

```
SQL>SELECT Empno, Ename, Job, HireDate, Sal FROM Emp
2 WHERE &1 = &2;

Enter value for 1: Deptno
Enter value for 2: 20
old   2: WHERE &1 = &2
new   2: WHERE Deptno = 20

EMPNO ENAME      JOB         HIREDATE       SAL
---------- ----------  ---------  ----------------  ----------
7369      SMITH       CLERK      17-DEC-80         800
7566      JONES       MANAGE     02-APR-81         2975
7788      SCOTT       ANALYST    19-APR-87         3000
7876      ADAMS       CLERK      23-MAY-87         1100
7902      FORD        ANALYST    03-DEC-81         3000
```

If you save the above SQL statement as a script file, you can submit the substitution variable values while invoking the script. Here is an example of saving and executing the script file.

```
SQL> save Demo_ex2
Created file Demo_ex2.sql
SQL> START demo_ex2 Deptno 10
old   2: WHERE &1 = &2
new   2: WHERE Deptno = 10
```

EMPNO	ENAME	JOB	HIREDATE	SAL
7782	CLARK	MANAGER	09-JUN-81	5981.45
7839	KING	PRESIDENT	17-NOV-81	12207.04
7934	MILLER	CLERK	23-JAN-82	3173.83

Self-Evaluation Test

Answer the following questions and then compare them to those given at the end of this chapter:

1. The _____ statement is the most popular SQL statement to query a table.

2. In Oracle, the _____ clause is used to prevent the selection of duplicate rows in a table.

3. The _____ operators are used to compare one expression with another.

4. The _____ clause is used to select, delete, or update only those rows in which the expression evaluates to true.

5. The _____ operator is used to compare the character string with the matching pattern.

6. The **SELECT** statement is used to retrieve or view the data from one or more tables. (T/F)

7. You can define the **WHERE** clause with only one condition. (T/F)

8. The **BETWEEN** operator is used to retrieve rows that fall within a specified range. (T/F)

9. The **IN** operator is used to retrieve rows based on the multiple value conditions. (T/F)

10. The **DEFINE** command is used to define a substitution variable in SQL *Plus. (T/F)

11. Which of the following is not a logical operator?

    (a) **AND** (b) **OR**
    (c) **NOT** (d) **IN**

12. Which of the following clause is used to filter the data from the database?

    (a) **WHERE** (b) **DESC**
    (c) **GROUP BY** (d) **ORDER BY**

13. Which of the following operators is used to combine the results from two or more queries into a single result.

    (a) **IN** (b) **SET**
    (c) **LIKE** (d) All of these

14. Which of the following clauses is used to arrange the data retrieved from a table into sorted order?

    (a) **HAVING** (b) **GROUP BY**
    (c) **ORDER BY** (d) **WHERE**

15. Which of the following is the aggregate function?

    (a) **SUM** (b) **COUNT**
    (c) Both (a) and (b) (d) None of these

Review Questions

Answer the following questions:

1. You can define the _____ clause with multiple conditions.

2. The _____ operator joins two or more than two conditions, and ensures that the rows satisfying the conditions are selected.

3. The _____ operator joins two or more than two conditions, and ensures that the rows satisfying any one of the conditions are selected.

4. The _____ operator joins the result set of the two **SELECT** statements.

5. The _____ operator returns distinct rows retrieved by either of the queries.

6. Set operators are used to combine the results from two or more queries into a single result. (T/F)

7. The **UNION** operator returns the difference between two sets. (T/F)

8. Set operators are not used with the **SELECT** statements containing the **TABLE** collection expressions. (T/F)

9. The **GROUP BY** clause is used in the **SELECT** statement to collect data across multiple records and group the results by one or more columns. (T/F)

10. The **LEFT OUTER JOIN** returns all rows of the first table and only those rows from the second table that follow the join condition. (T/F)

11. Which of the following operators are used to compare one expression with another expression?

    (a) Arithmetic
    (b) Logical
    (c) Comparison
    (d) None of these

12. Which of the following is the correct syntax for using the **AND** operator:

    (a) SELECT Column1, Column2.....
        From Table
        WHERE Condition1 AND Condition2;
    (b) SELECT Column1, Column2.....
        FROM Table
        WHERE Condition1 & Condition2:
    (c) SELECT Column1, Column2.....
        FROM Table
        WHERE Condition1 && Condition2.
    (d) None of these.

13. Which of the following keywords belongs to the SET operators?

    (a) **UNION**
    (b) **MINUS**
    (c) Both (a) and (b)
    (d) None of these

14. Which of the following operators cannot be applied on the columns of a data type?

    (a) **BLOB**
    (b) **BFILE**
    (c) Both (a) and (b)
    (d) None of these

15. Which of the following joins returns a null value in place of the rows which do not match the join condition from the other table.

    (a) **INNER JOIN**
    (b) **OUTER JOIN**
    (c) **LEFT OUTER JOIN**
    (d) **RIGHT OUTER JOIN**

Exercises

Exercise 1

Write a query using the **INTERSECT** command.

Exercise 2

Write a query to return all distinct rows retrieved by either of the queries using the **UNION** operator.

Exercise 3

Write a query to display the names of those employees who earn the lowest salary in a department.

Answers to Self-Evaluation Test

1. SELECT, **2. DISTINCT**, **3.** comparison, **4. WHERE**, **5. LIKE**, **6.** T, **7.** F, **8.** Y, **9.** T, **10.** T, **11.** c, **12.** a, **13.** b, **14.** c, **15.** d.