

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*
- *Work with Hierarchical Query Operators (PRIOR and CONNECT_BY_ROOT)*
- *Understand the Set operators (UNION, UNION ALL, INTERSECT, and MINUS)*
- *Understand the operator precedence*
- *Understand the CASE expression*
- *Understand Subqueries*
- *Apply Joins*
- *Work with the table and column aliases*
- *Understand pivot and unpivot queries*

INTRODUCTION

Data retrieval is the process of fetching data from database. You can retrieve data from database using the **SELECT** statement. To retrieve desired data from the database, you need a set of criteria and need to perform some operations on data through queries.

In this chapter, you will learn how to retrieve the desired data from database. You will also learn about SQL Operators, Operator precedence, Subqueries, pivot and unpivot queries, and JOINS. Additionally, you will learn about working with 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:

Execute the following query in SQL Worksheet to get the record of all employees, as shown in Figure 3-1.

```
SELECT * FROM EMPLOYEES;
```

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

The above SQL query retrieves all information contained within the **EMPLOYEES** table. Note that the asterisk is used as a wildcard in SQL. Literally, it means “Select all records from 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 for which you want to retrieve data. **Table_Name** is name of the table from which you want to retrieve data.

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
1	100	Steven	King	SKING	515.123.4567	17-06-03	AD_PRES	24000	(null)	(null)	90
2	101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-09-05	AD_VP	17000	(null)	100	90
3	102	Lex	De Haan	LDEHAAN	515.123.4569	13-01-01	AD_VP	17000	(null)	100	90
4	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-01-06	IT_PROG	9000	(null)	102	60
5	104	Bruce	Ernat	BERNST	590.423.4568	21-05-07	IT_PROG	6000	(null)	103	60
6	105	David	Austin	DAUSTIN	590.423.4569	25-06-05	IT_PROG	4800	(null)	103	60
7	106	Valli	Pataballa	VPATABAL	590.423.4560	05-02-06	IT_PROG	4800	(null)	103	60
8	107	Diana	Lorentz	DLORENTZ	590.423.5567	07-02-07	IT_PROG	4200	(null)	103	60
9	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-08-02	FI_MGR	12008	(null)	101	100
10	109	Daniel	Faviet	DFAVIET	515.124.4169	16-08-02	FI_ACCOUNT	9000	(null)	108	100
11	110	John	Chen	JCHEN	515.124.4269	28-09-05	FI_ACCOUNT	8200	(null)	108	100
12	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-09-05	FI_ACCOUNT	7700	(null)	108	100
13	112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-03-06	FI_ACCOUNT	7800	(null)	108	100
14	113	Luis	Popp	LPOPP	515.124.4567	07-12-07	FI_ACCOUNT	6900	(null)	108	100
15	114	Den	Raphaely	DRAPHEAL	515.127.4561	07-12-02	PU_MAN	11000	(null)	100	30
16	115	Alexander	Khao	AKHAO	515.127.4562	18-05-03	PU_CLERK	3100	(null)	114	30
17	116	Shelli	Baida	SBAIDA	515.127.4563	24-12-05	PU_CLERK	2900	(null)	114	30
18	117	Sigal	Tobias	STOBIAS	515.127.4564	24-07-05	PU_CLERK	2800	(null)	114	30
19	118	Guy	Himuro	GHIHURO	515.127.4565	15-11-06	PU_CLERK	2600	(null)	114	30
20	119	Karen	Colmenares	KCOLMENA	515.127.4566	10-08-07	PU_CLERK	2500	(null)	114	30
21	120	Matthew	Weiss	MWEISS	650.123.1234	18-07-04	ST_MAN	8000	(null)	100	50

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

For example:

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

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME FROM EMPLOYEES;
```

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 for 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 Worksheet and then execute it. The output of the query will be displayed, as shown in Figure 3-2.

```
SELECT JOB_ID FROM EMPLOYEES;
```

The screenshot shows an Oracle SQL Worksheet with a 'Query Builder' tab. The query entered is `SELECT JOB_ID FROM EMPLOYEES;`. Below the query, the 'Query Result' window displays the output. It shows a table with one column, 'JOB_ID', and 27 rows. The rows contain the following job IDs: FI_ACCOUNT (rows 7-11), FI_MGR (row 12), HR_REP (row 13), IT_PROG (rows 14-18), MK_MAN (row 19), MK_REP (row 20), PR_REP (row 21), PU_CLERK (rows 22-26), and PU_MAN (row 27). The status bar indicates 'Fetched 50 rows in 0.006 seconds'.

	JOB_ID
7	FI_ACCOUNT
8	FI_ACCOUNT
9	FI_ACCOUNT
10	FI_ACCOUNT
11	FI_ACCOUNT
12	FI_MGR
13	HR_REP
14	IT_PROG
15	IT_PROG
16	IT_PROG
17	IT_PROG
18	IT_PROG
19	MK_MAN
20	MK_REP
21	PR_REP
22	PU_CLERK
23	PU_CLERK
24	PU_CLERK
25	PU_CLERK
26	PU_CLERK
27	PU_MAN

Figure 3-2 Output with duplicate rows

This query will retrieve a list of job ids. Notice that the job ids **FI_ACCOUNT**, **IT_PROG**, and **PU_CLERK** appear more than once. Now, if you want to retrieve the list of different job ids with no job ids 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_ID FROM EMPLOYEES;
```

The above query will retrieve all distinct job ids from the **EMPLOYEES** table.

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 screenshot shows a SQL Worksheet interface. At the top, there are tabs for 'Worksheet' and 'Query Builder'. The 'Query Builder' tab is active, displaying the SQL query: `SELECT DISTINCT JOB_ID FROM EMPLOYEES;`. Below the query, there is a 'Query Result' section. It shows a table with one column, 'JOB_ID', and 19 rows of data. The status bar indicates 'All Rows Fetched: 19 in 0.005 seconds'.

JOB_ID
1 AC_ACCOUNT
2 AC_MGR
3 AD_ASST
4 AD_PRES
5 AD_VP
6 FI_ACCOUNT
7 FI_MGR
8 HR_REP
9 IT_PROG
10 MK_MAN
11 MK_REP
12 PR_REP
13 PU_CLERK
14 PU_MAN
15 SA_MAN
16 SA_REP
17 SH_CLERK
18 ST_CLERK
19 ST_MAN

Figure 3-3 Output with distinct rows

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 Worksheet and then execute them. The output of the query will be displayed, as shown in Figure 3-4.

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, HIRE_DATE, SALARY,
JOB_ID FROM EMPLOYEES WHERE JOB_ID= 'IT_PROG';
```

In the above SQL statement, the **WHERE** clause will filter the data from the **EMPLOYEES** table. The above SQL statement will return all rows having the job id **IT_PROG**.

The screenshot shows the Oracle SQL Developer interface. At the top, there's a 'Worksheet' tab and a 'Query Builder' tab. The SQL text area contains the following query:

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, HIRE_DATE, SALARY, JOB_ID FROM EMPLOYEES
WHERE JOB_ID = 'IT_PROG';
```

Below the SQL text area, there's a 'Query Result' tab. It shows the results of the query, indicating that all rows were fetched in 0.004 seconds. The results are displayed in a table with the following columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, HIRE_DATE, SALARY, and JOB_ID. There are 5 rows of data.

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE	SALARY	JOB_ID
1	103	Alexander	Hunold	03-01-06	9000	IT_PROG
2	104	Bruce	Ernst	21-05-07	6000	IT_PROG
3	105	David	Austin	25-06-05	4800	IT_PROG
4	106	Valli	Pataballa	05-02-06	4800	IT_PROG
5	107	Diana	Lorentz	07-02-07	4200	IT_PROG

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

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 EMPLOYEES.FIRST_NAME, DEPARTMENTS.DEPARTMENT_NAME
FROM EMPLOYEES RIGHT OUTER JOIN DEPARTMENTS
ON EMPLOYEES.DEPARTMENT_ID=DEPARTMENTS.DEPARTMENT_ID;
```

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

```
SELECT E.FIRST_NAME, D.DEPARTMENT_NAME
FROM EMPLOYEES E RIGHT OUTER JOIN DEPARTMENTS D
ON E.DEPARTMENT_ID=D.DEPARTMENT_ID;
```

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

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 EMPLOYEE_ID, FIRST_NAME
FROM EMPLOYEES;
```

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

```
SELECT EMPLOYEE_ID "ID", FIRST_NAME "NAME"
FROM EMPLOYEES;
```

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

```
SELECT EMPLOYEE_ID "ID", FIRST_NAME || ' ' || LAST_NAME "Name",
SALARY "Basic Salary", SALARY + NVL(COMMISSION_PCT, 0) "Net Salary"
FROM EMPLOYEES WHERE SALARY >= 12000;
```

The above query will display four columns from the table **EMPLOYEES**. The first column will have the heading **ID**, the second column will have the heading **Name**, third and fourth columns will have headings **Basic Salary** and **Net Salary**. The output of the above query is shown in Figure 3-5.

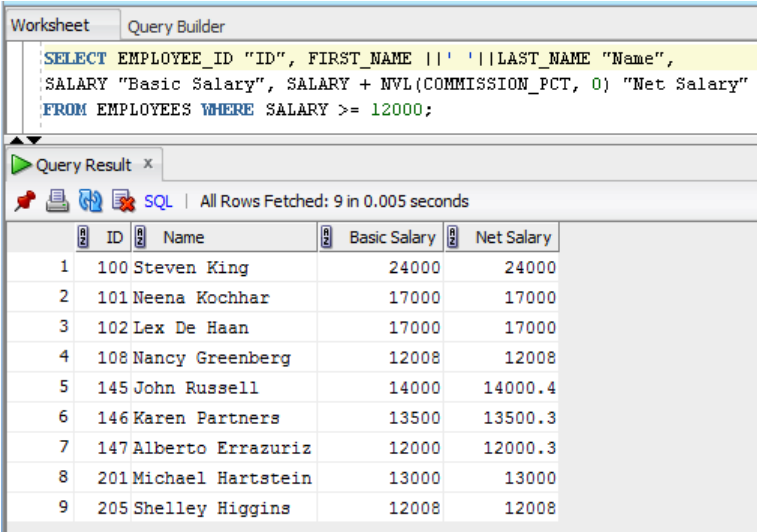


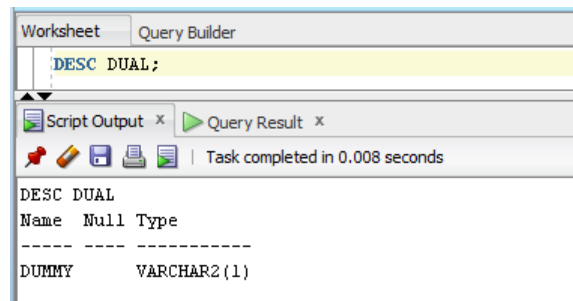
Figure 3-5 Retrieving data with column alias names



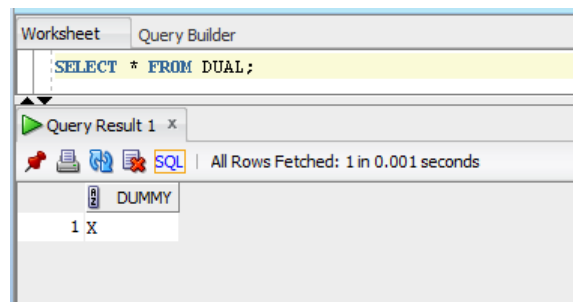
Note
The **NVL** function will be discussed in later chapters.

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, as shown in Figure 3-6. The table has a single row with a value of **X** (here **X** can be any value), as shown in Figure 3-7. The owner of the dual table is **SYS** but it can be accessed by every user in the Oracle database.



*Figure 3-6 Structure of **DUAL** table*

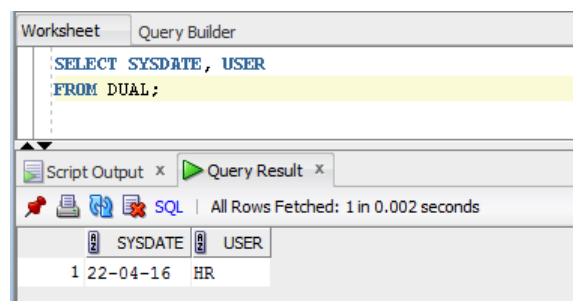


*Figure 3-7 Row of **DUAL** table*

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

```
SELECT SYSDATE, USER
FROM DUAL;
```

In the above example, the **SELECT** statement query will return the system date and the name of current user, as shown in Figure 3-8. The **SYSDATE** function is used to return the system date and is discussed in the later chapters.



*Figure 3-8 Selecting **SYSDATE** and **USER** from the **DUAL** table*

SQL OPERATORS

An operator is a symbol or character that is capable of manipulating individual data items and then returning a result. The data item on which an operator operates is called an operand. An operator can operate on a single operand or two operands. Operators that operate on single operand are called Unary operators and the operators that operate on two operators are called Binary operators. The following operators are supported by Oracle:

- 1. Arithmetic Operators
- 2. Concatenation Operators
- 3. Hierarchical Query Operators
- 4. Comparison Operators
- 5. Logical Operators
- 6. Other Operators
- 7. Set Operators

These operators are discussed next.

Arithmetic Operators

Oracle database uses arithmetic operator to perform arithmetic operations on one or more numeric values. Some of the arithmetic operators are also used with datetime and interval operations. The arithmetic operators and their usage are discussed in Table 3-1.

Table 3-1 The Arithmetic operators and their description

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.

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;
```

Output:

```
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;
```

Output:

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

The following query adds specified number of days to SYSDATE:

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

Output

```
SYSDATE          RESULT_DATE
-----
22-04-16         02-05-16
```



Note

The output of above query will depend on the current system date.

Concatenation Operators

The concatenation operators allow you to combine two or more characters or strings, columns together into one expression. If any of the concatenation values is NULL, Oracle treats it as zero length character string and returns the string having a value. In Oracle, two solid vertical bars || are used as concatenation operator.

For example:

```
SELECT 'CADCIM' || ' ' || 'TECHNOLOGIES' "COMPANY"
FROM DUAL;
```

Output:

```
COMPANY
-----
CADCIM TECHNOLOGIES
```

Hierarchical Query Operators

Hierarchical query operators are used on the tables which contain hierarchical data. Hierarchical data is a parent-child relationship of data within the same table or view. There are two hierarchical query operators **PRIOR** and **CONNECT_BY_ROOT**.

PRIOR

PRIOR is a unary operator which is used with **CONNECT BY** clause in hierarchical queries. It can exist on either side of equality condition of the clause. It is mostly used to compare column values with the equality operator. Due to **PRIOR** operator, the direction of hierarchy flow is decided on the basis of the **CONNECT BY** clause condition.

The syntax for using **PRIOR** operator is as follows:

```
SELECT
FROM
START WITH
CONNECT BY [PRIOR] condition
```

In the above syntax, the **START WITH** clause specifies a condition that identifies the rows to be used as the root of a hierarchical query.

For example:

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, MANAGER_ID,
       LPAD(LEVEL, 5*(LEVEL)) "LEVEL"
FROM EMPLOYEES WHERE EMPLOYEE_ID<115
START WITH EMPLOYEE_ID = 100
CONNECT BY PRIOR EMPLOYEE_ID = MANAGER_ID
ORDER SIBLINGS BY FIRST_NAME;
```

The above SQL query will show the hierarchical relationship of employees and managers in an organization. The **SIBLINGS** keyword used in **ORDER BY** clause preserves any ordering within the hierarchy. The **LEVEL** is the pseudocolumn which returns 1 for a root row and 2 for a child row, and so on. The output of above query is shown in Figure 3-9.

The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying a SQL query that uses the **PRIOR** operator to traverse the employee hierarchy. The query is as follows:

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, MANAGER_ID, LPAD(LEVEL, 5*(LEVEL)) "LEVEL"
FROM EMPLOYEES WHERE EMPLOYEE_ID<115
START WITH EMPLOYEE_ID = 100
CONNECT BY PRIOR EMPLOYEE_ID = MANAGER_ID
ORDER SIBLINGS BY FIRST_NAME;
```

Below the query, the 'Query Result' tab shows the output of the query. It indicates that 15 rows were fetched in 0.007 seconds. The results are displayed in a table with the following columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, MANAGER_ID, and LEVEL. The LEVEL column is formatted with LPAD to show the hierarchy level.

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	MANAGER_ID	LEVEL
1	100	Steven	King	(null)	1
2	114	Den	Raphaely	100	2
3	102	Lex	De Haan	100	2
4	103	Alexander	Hunold	102	3
5	104	Bruce	Ernst	103	4
6	105	David	Austin	103	4
7	107	Diana	Lorentz	103	4
8	106	Valli	Pataballa	103	4
9	101	Neena	Kochhar	100	2
10	108	Nancy	Greenberg	101	3
11	109	Daniel	Faviet	108	4
12	111	Ismael	Sciarra	108	4
13	110	John	Chen	108	4
14	112	Jose Manuel	Urman	108	4
15	113	Luis	Popp	108	4

Figure 3-9 Using the **PRIOR** operator with the **SELECT** statement

CONNECT_BY_ROOT

CONNECT_BY_ROOT is a unary operator which is used in hierarchical queries. It enhances the functionality of **CONNECT BY [PRIOR]** condition. Oracle returns the column values from the root node associated with row of the column specified with the **CONNECT_BY_ROOT** operator.

For example:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "EMPLOYEE",
MANAGER_ID, CONNECT_BY_ROOT FIRST_NAME "MANAGER",
LEVEL-1 "LEVEL", SYS_CONNECT_BY_PATH(FIRST_NAME, '/') "HIERARCHY"
FROM EMPLOYEES WHERE LEVEL > 1 and DEPARTMENT_ID = 60
CONNECT BY PRIOR EMPLOYEE_ID = MANAGER_ID
ORDER BY "EMPLOYEE", "MANAGER", "LEVEL", "HIERARCHY";
```

The above example returns the first name of each employee in department 60, each manager at the level upper than the employee in the hierarchy, the number of levels between manager and employee, and the path between the two. The output of the above query is shown in Figure 3-10.

The screenshot shows a SQL Query Builder window with a query that uses the `CONNECT_BY_ROOT` operator to retrieve hierarchical data from the `EMPLOYEES` table. The query filters for `DEPARTMENT_ID = 60` and orders the results by employee, manager, level, and hierarchy. Below the query, the 'Script Output' tab displays the results in a table with 14 rows.

	EMPLOYEE_ID	EMPLOYEE	MANAGER_ID	MANAGER	LEVEL	HIERARCHY
1	103	Alexander Hunold	102	Lex	1	/Lex/Alexander
2	103	Alexander Hunold	102	Steven	2	/Steven/Lex/Alexander
3	104	Bruce Ernst	103	Alexander	1	/Alexander/Bruce
4	104	Bruce Ernst	103	Lex	2	/Lex/Alexander/Bruce
5	104	Bruce Ernst	103	Steven	3	/Steven/Lex/Alexander/Bruce
6	105	David Austin	103	Alexander	1	/Alexander/David
7	105	David Austin	103	Lex	2	/Lex/Alexander/David
8	105	David Austin	103	Steven	3	/Steven/Lex/Alexander/David
9	107	Diana Lorentz	103	Alexander	1	/Alexander/Diana
10	107	Diana Lorentz	103	Lex	2	/Lex/Alexander/Diana
11	107	Diana Lorentz	103	Steven	3	/Steven/Lex/Alexander/Diana
12	106	Valli Pataballa	103	Alexander	1	/Alexander/Valli
13	106	Valli Pataballa	103	Lex	2	/Lex/Alexander/Valli
14	106	Valli Pataballa	103	Steven	3	/Steven/Lex/Alexander/Valli

Figure 3-10 Using the `CONNECT_BY_ROOT` operator with the `SELECT` statement

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 value or the result of expression on both sides of the operator is equal, the condition will be TRUE.

For example:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY,
DEPARTMENT_ID FROM EMPLOYEES WHERE DEPARTMENT_ID=30;
```

In the above example, the SQL query will return all those records of the **EMPLOYEES** table in which the department number is 30. The output of above query is shown in Figure 3-11.

The screenshot shows the Oracle SQL Developer Query Builder interface. The SQL statement entered is: `SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, DEPARTMENT_ID FROM EMPLOYEES WHERE DEPARTMENT_ID=30;`. The query result is displayed in a table with 6 rows and 4 columns: EMPLOYEE_ID, ENAME, SALARY, and DEPARTMENT_ID. The status bar indicates "All Rows Fetched: 6 in 0.003 seconds".

	EMPLOYEE_ID	ENAME	SALARY	DEPARTMENT_ID
1	114	Den Raphaely	11000	30
2	115	Alexander Khoo	3100	30
3	116	Shelli Baida	2900	30
4	117	Sigal Tobias	2800	30
5	118	Guy Himuro	2600	30
6	119	Karen Colmenares	2500	30

Figure 3-11 Using the Equal operator with the **SELECT** statement

!=, <>, 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 * FROM DEPARTMENTS WHERE LOCATION_ID <> 1700;
```

In the above example, the SQL query will return all those records of the **DEPARTMENTS** table, in which the location id is not 1700. The output of the above query is shown in Figure 3-12.

The screenshot shows the Oracle SQL Developer Query Builder interface. The SQL statement entered is: `SELECT * FROM DEPARTMENTS WHERE LOCATION_ID <> 1700;`. The query result is displayed in a table with 6 rows and 4 columns: DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID. The status bar indicates "All Rows Fetched: 6 in 0.003 seconds".

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	20	Marketing	201	1800
2	40	Human Resources	203	2400
3	50	Shipping	121	1500
4	60	IT	103	1400
5	70	Public Relations	204	2700
6	80	Sales	145	2500

Figure 3-12 Using the Not Equal to operator with the **SELECT** statement

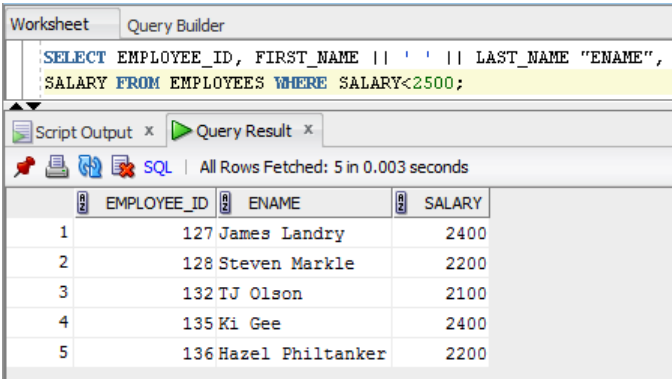
< (Less Than)

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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY
FROM EMPLOYEES WHERE SALARY<2500;
```

In the above example, the SQL query will return all those records from the **EMPLOYEES** table in which the salary is less than **2500**. The output of the above query is shown in Figure 3-13.



The screenshot shows the SQL Developer interface. The 'Query Builder' tab is active, displaying the following SQL query: `SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY FROM EMPLOYEES WHERE SALARY<2500;`. Below the query, the 'Query Result' tab shows the output of the query. The results are displayed in a table with three columns: EMPLOYEE_ID, ENAME, and SALARY. The table contains five rows of data.

EMPLOYEE_ID	ENAME	SALARY
127	James Landry	2400
128	Steven Markle	2200
132	TJ Olson	2100
135	Ki Gee	2400
136	Hazel Philtanker	2200

Figure 3-13 Using the Less Than operator with the **SELECT** statement

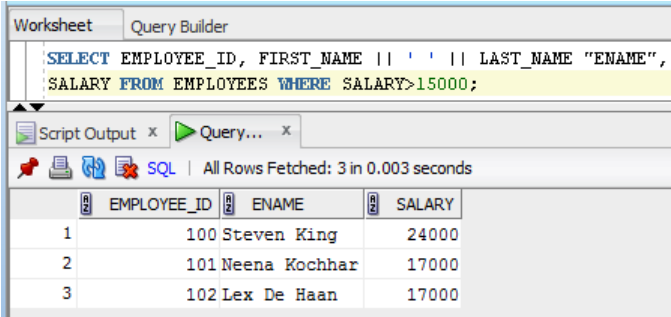
> (Greater Than)

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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY
FROM EMPLOYEES WHERE SALARY>15000;
```

In the above example, the SQL query will return all those records from the **EMPLOYEES** table in which the salary is greater than **15000**. The output of the above query is shown in Figure 3-14.



The screenshot shows the SQL Developer interface. The 'Query Builder' tab is active, displaying the following SQL query: `SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY FROM EMPLOYEES WHERE SALARY>15000;`. Below the query, the 'Query Result' tab shows the output of the query. The results are displayed in a table with three columns: EMPLOYEE_ID, ENAME, and SALARY. The table contains three rows of data.

EMPLOYEE_ID	ENAME	SALARY
100	Steven King	24000
101	Neena Kochhar	17000
102	Lex De Haan	17000

Figure 3-14 Using the Greater Than operator with the **SELECT** statement

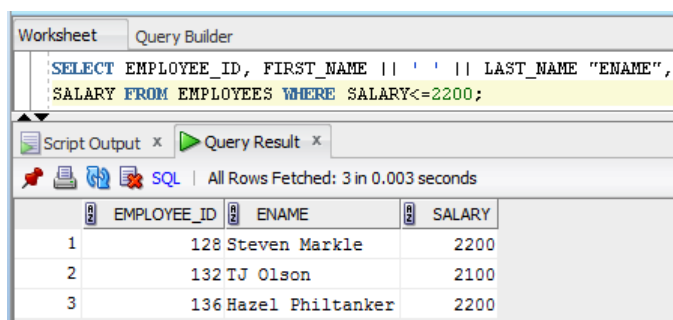
<= (Less Than or Equal to)

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 than or Equal to) operator will evaluate to TRUE.

For example:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY
FROM EMPLOYEES WHERE SALARY<=2200;
```

In the above example, the SQL query returns all those records of the **EMPLOYEES** table in which the salary is less than or equal to **2200**. The output of the above query is shown in Figure 3-15.



The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying the following SQL query: `SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY FROM EMPLOYEES WHERE SALARY<=2200;`. Below the query, the 'Query Result' tab shows the output of the query. It indicates 'All Rows Fetched: 3 in 0.003 seconds'. The results are displayed in a table with three columns: EMPLOYEE_ID, ENAME, and SALARY. The data rows are: 128 Steven Markle (2200), 132 TJ Olson (2100), and 136 Hazel Philtanker (2200).

EMPLOYEE_ID	ENAME	SALARY
128	Steven Markle	2200
132	TJ Olson	2100
136	Hazel Philtanker	2200

Figure 3-15 Using the Less Than or Equal to operator with the **SELECT** statement

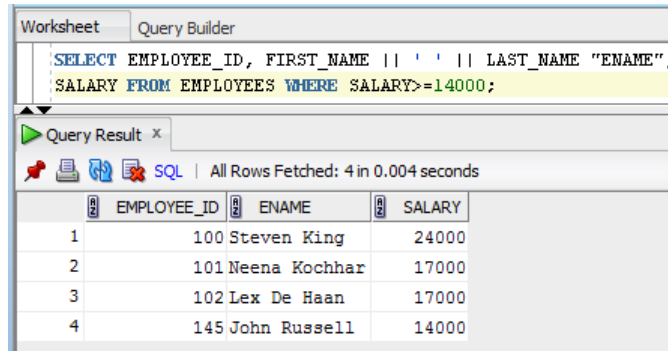
>= (Greater Than or Equal To)

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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY
FROM EMPLOYEES WHERE SALARY>=14000;
```

In the above example, the SQL query will return all those records of the **EMPLOYEES** table in which the salary is either equal to or more than **14000**. The output of the above query is shown in Figure 3-16.



Worksheet Query Builder

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
SALARY FROM EMPLOYEES WHERE SALARY >= 14000;
```

Query Result x

All Rows Fetched: 4 in 0.004 seconds

	EMPLOYEE_ID	ENAME	SALARY
1	100	Steven King	24000
2	101	Neena Kochhar	17000
3	102	Lex De Haan	17000
4	145	John Russell	14000

Figure 3-16 Using the Greater Than or Equal to operator with **SELECT** statement

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 >=.

If **ANY** or **SOME** comparison operator is followed by a list of values, Oracle optimizer expands the condition to all the values of the list together with the **OR** operator.

For example:

```
SALARY > ANY(2000, 3000)
```

transformed to

```
SALARY > 2000 OR SALARY > 3000
```

Also, if **ANY** or **SOME** operator is followed by the subquery, Oracle optimizer transforms it into a condition containing the **EXISTS** operator and a subquery.

For example:

```
SALARY1 > ANY(SELECT SALARY FROM EMPLOYEES
WHERE DEPARTMENT_ID=60)
```

transformed to

```
EXISTS(SELECT SALARY FROM EMPLOYEES
WHERE DEPARTMENT_ID=60 AND SALARY1>SALARY)
```

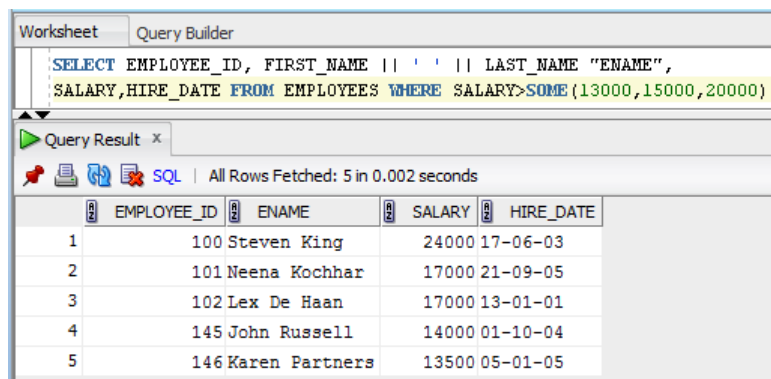
The following example will illustrate the use of **ANY** and **SOME** operators with the comparison operator >(greater than):

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
       SALARY, HIRE_DATE FROM EMPLOYEES WHERE SALARY > SOME (13000, 15000, 20000);
```

Or

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
       SALARY, HIRE_DATE FROM EMPLOYEES WHERE SALARY > ANY (13000, 15000, 20000);
```

In this example, the SQL query will return all those records of the **EMPLOYEES** table in which the salary of employee is greater than any of values in the list (13000, 15000, 20000). The output of the above query is shown in Figure 3-17.



	EMPLOYEE_ID	ENAME	SALARY	HIRE_DATE
1	100	Steven King	24000	17-06-03
2	101	Neena Kochhar	17000	21-09-05
3	102	Lex De Haan	17000	13-01-01
4	145	John Russell	14000	01-10-04
5	146	Karen Partners	13500	05-01-05

Figure 3-17 Using the **SOME** operator with the **SELECT** statement

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

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
       SALARY, HIRE_DATE FROM EMPLOYEES WHERE SALARY = ANY (13000, 15000, 20000);
```

Or

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
       SALARY, HIRE_DATE FROM EMPLOYEES WHERE SALARY = SOME (13000, 15000, 20000);
```

In the above example, the SQL query will return all those records of the **EMPLOYEES** table in which the salary is equal to the values in the list (13000, 15000, 20000).



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 **>=**.

If the **ALL** comparison operator is followed by a list of values, Oracle optimizer expands the condition to all the values of the list together with the **AND** operator.

For example:

```
SALARY > ALL(2000, 3000)
```

transformed to

```
SALARY > 2000 AND SALARY > 3000
```

Also, if the **ALL** operator is followed by a subquery, Oracle optimizer transforms it into a condition that uses **ANY** comparison operator and a complementary comparison operator including subquery.

For example:

```
SALARY1 > ALL(SELECT SALARY FROM EMPLOYEES  
WHERE DEPARTMENT_ID=60)
```

transformed to

```
NOT (SALARY1 <= ANY(SELECT SALARY FROM EMPLOYEES  
WHERE DEPARTMENT_ID = 60))
```

After further transforming the **ANY** operator above condition will look like:

```
NOT EXISTS (SELECT SALARY FROM EMPLOYEES  
WHERE DEPARTMENT_ID=60 AND SALARY1 <= SALARY)
```

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

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",  
SALARY, HIRE_DATE FROM EMPLOYEES WHERE SALARY > ALL(8000, 10000, 13000);
```

In this example, the SQL query will return all those records of the **EMPLOYEES** table in which the salary of employee is greater than each of the values in the list (8000, 10000, 13000). The output of the above query is shown in Figure 3-18.

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

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",  
SALARY, HIRE_DATE FROM EMPLOYEES WHERE SALARY >= ALL(8000, 10000, 13000);
```

In the above example, the SQL query will return all those records of the **EMPLOYEES** table in which the salary of employee is greater than or equal to the values in the list (8000, 1000, 13000).

The screenshot shows the Oracle SQL Developer Query Builder interface. The SQL statement entered is: `SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, HIRE_DATE FROM EMPLOYEES WHERE SALARY > ALL (8000, 10000, 13000);`. The results are displayed in a table with 5 rows.

EMPLOYEE_ID	ENAME	SALARY	HIRE_DATE
1	100 Steven King	24000	17-06-03
2	101 Neena Kochhar	17000	21-09-05
3	102 Lex De Haan	17000	13-01-01
4	145 John Russell	14000	01-10-04
5	146 Karen Partners	13500	05-01-05

Figure 3-18 Using the **ALL** operator with the **SELECT** statement

Logical Operators

The logical operators are used to compare two or more conditions to produce 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 EMPLOYEES
WHERE NOT (JOB_ID IS NULL);
```

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

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

```
SELECT * FROM EMPLOYEES
WHERE NOT (SALARY BETWEEN 1000 AND 2000);
```

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

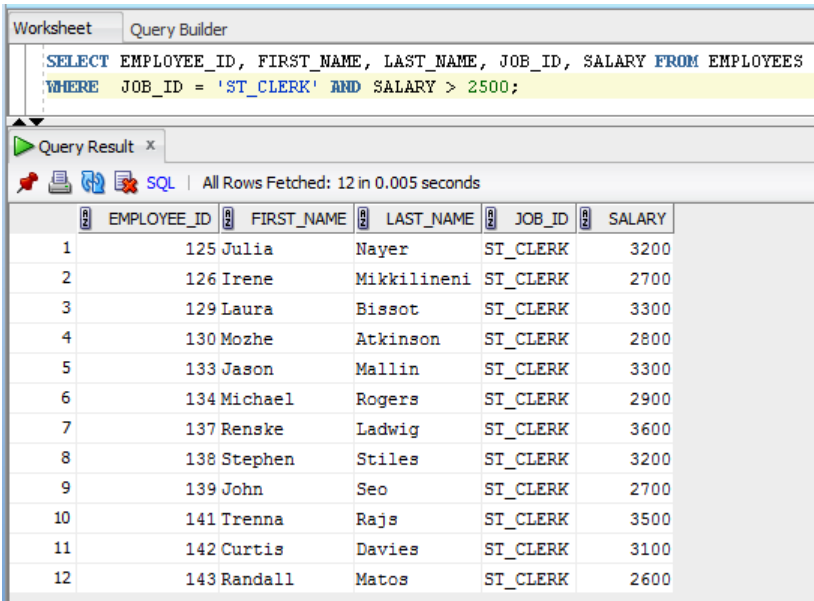
AND

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 EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY
FROM EMPLOYEES WHERE JOB_ID = 'ST_CLERK' AND SALARY > 2500;
```

The above query will return all those records of the **EMPLOYEES** table in which both the conditions, **JOB_ID = 'ST_CLERK'** and **SALARY > 2500**, return TRUE, as shown in Figure 3-19.



The screenshot shows a 'Query Builder' window with the following SQL query: `SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY FROM EMPLOYEES WHERE JOB_ID = 'ST_CLERK' AND SALARY > 2500;`. Below the query, the 'Query Result' tab displays 12 rows of data. The status bar indicates 'All Rows Fetched: 12 in 0.005 seconds'.

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	125	Julia	Nayer	ST_CLERK	3200
2	126	Irene	Mikkilineni	ST_CLERK	2700
3	129	Laura	Bissot	ST_CLERK	3300
4	130	Mozhe	Atkinson	ST_CLERK	2800
5	133	Jason	Mallin	ST_CLERK	3300
6	134	Michael	Rogers	ST_CLERK	2900
7	137	Renske	Ladwig	ST_CLERK	3600
8	138	Stephen	Stiles	ST_CLERK	3200
9	139	John	Seo	ST_CLERK	2700
10	141	Trenna	Rajs	ST_CLERK	3500
11	142	Curtis	Davies	ST_CLERK	3100
12	143	Randall	Matos	ST_CLERK	2600

Figure 3-19 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 will 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 EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY
FROM EMPLOYEES WHERE JOB_ID = 'ST_CLERK' OR SALARY > 15000;
```

This above query will return all those records from the **EMPLOYEES** table in which one of the conditions, **JOB_ID = 'IT_PROG'** or **SALARY > 15000** is TRUE, as shown in Figure 3-20.

The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying a SQL query: `SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, SALARY FROM EMPLOYEES WHERE JOB_ID = 'IT_PROG' OR SALARY > 15000;`. Below the query, the 'Query Result' tab shows the results of the query. It indicates 'All Rows Fetched: 8 in 0.006 seconds'. The results are displayed in a table with 8 rows and 5 columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, JOB_ID, and SALARY.

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	SALARY
1	100	Steven	King	AD_PRES	24000
2	101	Neena	Kochhar	AD_VP	17000
3	102	Lex	De Haan	AD_VP	17000
4	103	Alexander	Hunold	IT_PROG	9000
5	104	Bruce	Ernst	IT_PROG	6000
6	105	David	Austin	IT_PROG	4800
7	106	Valli	Pataballa	IT_PROG	4800
8	107	Diana	Lorentz	IT_PROG	4200

Figure 3-20 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';
```

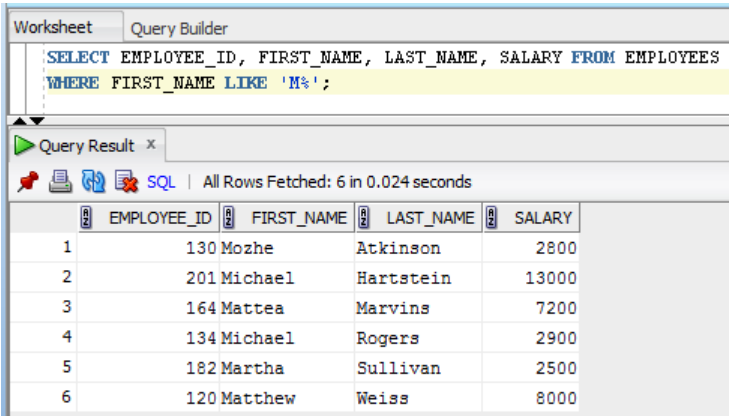
In the above syntax, **Char_String** is the pattern with which the **Column_Name** will be compared. The pattern is a value having the data type **CHAR** or **VARCHAR2** and contains the special matching pattern characters: percent sign (%) and underscore (_).

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

For example:

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY FROM EMPLOYEES
WHERE FIRST_NAME LIKE 'M%';
```

The above query will return all the rows of **EMPLOYEES** table where the **FIRST_NAME** starts with letter M. The output of the above query is shown in Figure 3-21.



The screenshot shows a SQL Worksheet interface. The 'Query Builder' tab is active, displaying the following SQL query: `SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY FROM EMPLOYEES WHERE FIRST_NAME LIKE 'M%';`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 6 rows and 4 columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, and SALARY. The data is as follows:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
1	130	Mozhe	Atkinson	2800
2	201	Michael	Hartstein	13000
3	164	Mattea	Marvins	7200
4	134	Michael	Rogers	2900
5	182	Martha	Sullivan	2500
6	120	Matthew	Weiss	8000

Figure 3-21 Query showing the use of the (%) percent sign with 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 Worksheet, enter the following SQL query to retrieve the rows in which the first name of employees begins with the letter **A**:

```
SELECT FIRST_NAME, SALARY, JOB_ID, COMMISSION_PCT, HIRE_DATE
FROM EMPLOYEES WHERE FIRST_NAME 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 employees whose first name begins with the character **A**.

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

```
SELECT FIRST_NAME, SALARY, JOB_ID, COMMISSION_PCT, HIRE_DATE
FROM EMPLOYEES WHERE FIRST_NAME LIKE '%en%';
```

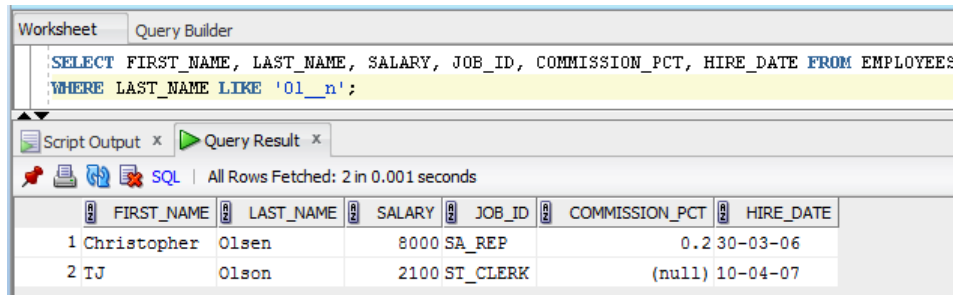
The above query will return all those employees whose first name contains the characters **en**. Note that this character set may appear anywhere in the name of the employees.

- 3. In SQL Worksheet, enter the following SQL query to retrieve the rows in which the last name of employee is similar to **Olsen** and **Olson**. In these names, the first two and the last

one character will remain the same. This can be done by using the underscore (_) with the **LIKE** operator.

```
SELECT FIRST_NAME, LAST_NAME, SALARY, JOB_ID, COMMISSION_PCT,
       HIRE_DATE FROM EMPLOYEES WHERE LAST_NAME LIKE 'Ol__n';
```

In the above example, the underscore sign (__) used twice between **Ol** and **s** in the **LIKE** operator represents any possible two characters that might appear between **Ol** and **s**. The output of the above query is shown in Figure 3-22.



The screenshot shows the Oracle SQL Worksheet interface. The 'Query Builder' tab is active, displaying the following SQL query:

```
SELECT FIRST_NAME, LAST_NAME, SALARY, JOB_ID, COMMISSION_PCT, HIRE_DATE FROM EMPLOYEES
WHERE LAST_NAME LIKE 'Ol__n';
```

Below the query, the 'Script Output' and 'Query Result' tabs are visible. The 'Query Result' tab shows the output of the query, indicating that 2 rows were fetched in 0.001 seconds. The results are displayed in a table with the following columns: FIRST_NAME, LAST_NAME, SALARY, JOB_ID, COMMISSION_PCT, and HIRE_DATE.

	FIRST_NAME	LAST_NAME	SALARY	JOB_ID	COMMISSION_PCT	HIRE_DATE
1	Christopher	Olsen	8000	SA_REP	0.2	30-03-06
2	TJ	Olsen	2100	ST_CLERK	(null)	10-04-07

*Figure 3-22 Query showing the use of the (__) underscore with 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 Worksheet, enter the following SQL query to retrieve the rows from the **EMPLOYEES** table having employee id between 190 and 200:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
       SALARY, JOB_ID, HIRE_DATE FROM EMPLOYEES
WHERE EMPLOYEE_ID BETWEEN 190 AND 200;
```

The above query will return the details of employees having employee id between 190 and 200, as shown in Figure 3-23.

The screenshot shows a SQL Query Builder window with the following SQL query:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
SALARY, JOB_ID, HIRE_DATE FROM EMPLOYEES WHERE EMPLOYEE_ID BETWEEN 190 AND 200;
```

The Query Result tab displays 11 rows of data:

	EMPLOYEE_ID	ENAME	SALARY	JOB_ID	HIRE_DATE
1	190	Timothy Gates	2900	SH_CLERK	11-07-06
2	191	Randall Perkins	2500	SH_CLERK	19-12-07
3	192	Sarah Bell	4000	SH_CLERK	04-02-04
4	193	Britney Everett	3900	SH_CLERK	03-03-05
5	194	Samuel McCain	3200	SH_CLERK	01-07-06
6	195	Vance Jones	2800	SH_CLERK	17-03-07
7	196	Alana Walsh	3100	SH_CLERK	24-04-06
8	197	Kevin Feeney	3000	SH_CLERK	23-05-06
9	198	Donald OConnell	2600	SH_CLERK	21-06-07
10	199	Douglas Grant	2600	SH_CLERK	13-01-08
11	200	Jennifer Whalen	4400	AD_ASST	17-09-03

Figure 3-23 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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
SALARY, JOB_ID, HIRE_DATE FROM EMPLOYEES
WHERE HIRE_DATE BETWEEN TO_DATE('25/09/2005', 'dd/mm/yy')
AND TO_DATE('25/01/2006', 'dd/mm/yy');
```

The preceding query will return all details of employees having hire date between Sept 25, 2005 and Jan 25, 2006, as shown in Figure 3-24.

The screenshot shows a SQL Query Builder window with the following SQL query:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, JOB_ID, HIRE_DATE FROM EMPLOYEES
WHERE HIRE_DATE BETWEEN TO_DATE('25/09/2005', 'dd/mm/yy') AND TO_DATE('25/01/2006', 'dd/mm/yy');
```

The Query Result tab displays 11 rows of data:

	EMPLOYEE_ID	ENAME	SALARY	JOB_ID	HIRE_DATE
1	103	Alexander Hunold	9000	IT_PROG	03-01-06
2	110	John Chen	8200	FI_ACCOUNT	28-09-05
3	111	Ismael Sciarra	7700	FI_ACCOUNT	30-09-05
4	116	Shelli Baida	2900	PU_CLERK	24-12-05
5	123	Shanta Vollman	6500	ST_MAN	10-10-05
6	130	Mozhe Atkinson	2800	ST_CLERK	30-10-05
7	138	Stephen Stiles	3200	ST_CLERK	26-10-05
8	160	Louise Doran	7500	SA_REP	15-12-05
9	162	Clara Vishney	10500	SA_REP	11-11-05
10	170	Taylor Fox	9600	SA_REP	24-01-06
11	180	Winston Taylor	3200	SH_CLERK	24-01-06

Figure 3-24 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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
       SALARY, JOB_ID, HIRE_DATE FROM EMPLOYEES
WHERE HIRE_DATE >= TO_DATE('25/09/2005', 'dd/mm/yy')
      AND HIRE_DATE <= TO_DATE('25/01/2006', 'dd/mm/yy');
```

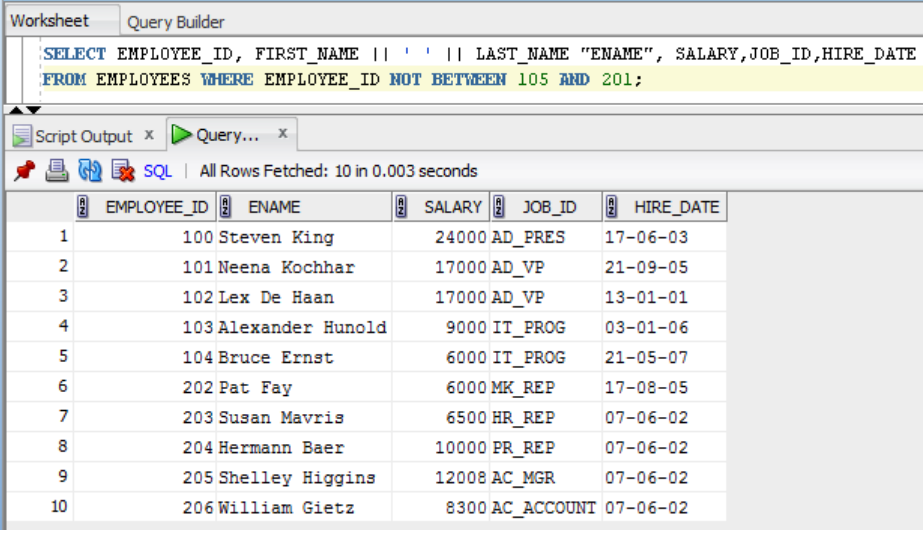
NOT BETWEEN

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 exist between the two given values or expressions.

For example:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
       SALARY, JOB_ID, HIRE_DATE FROM EMPLOYEES
WHERE EMPLOYEE_ID NOT BETWEEN 105 AND 201;
```

The above query will return all details of those employees whose **EMPLOYEE_ID** is not between 105 and 201, as shown in Figure 3-25.



The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying the following SQL query:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, JOB_ID, HIRE_DATE
FROM EMPLOYEES WHERE EMPLOYEE_ID NOT BETWEEN 105 AND 201;
```

Below the query editor, the 'Script Output' pane shows the results of the query. It indicates 'All Rows Fetched: 10 in 0.003 seconds'. The results are displayed in a table with the following columns: EMPLOYEE_ID, ENAME, SALARY, JOB_ID, and HIRE_DATE.

EMPLOYEE_ID	ENAME	SALARY	JOB_ID	HIRE_DATE
100	Steven King	24000	AD_PRES	17-06-03
101	Neena Kochhar	17000	AD_VP	21-09-05
102	Lex De Haan	17000	AD_VP	13-01-01
103	Alexander Hunold	9000	IT_PROG	03-01-06
104	Bruce Ernst	6000	IT_PROG	21-05-07
202	Pat Fay	6000	MK_REP	17-08-05
203	Susan Mavris	6500	HR_REP	07-06-02
204	Hermann Baer	10000	PR_REP	07-06-02
205	Shelley Higgins	12008	AC_MGR	07-06-02
206	William Gietz	8300	AC_ACCOUNT	07-06-02

*Figure 3-25 Query showing the use of the **NOT BETWEEN** operator*

The above query can be also written as:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
       SALARY, JOB_ID, HIRE_DATE FROM EMPLOYEES
WHERE EMPLOYEE_ID < 105 OR EMPLOYEE_ID > 201;
```

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 Worksheet, enter the following query to retrieve details of those employees whose employee numbers are 190, 195, and 200.

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
SALARY, JOB_ID, HIRE_DATE, MANAGER_ID FROM EMPLOYEES
WHERE EMPLOYEE_ID IN(190, 195, 200);
```

The list of values enclosed in the parenthesis is called an inlist. The above query has an inlist with three values (190, 195, 200). The above query will return the details of those employees whose employee number is same as in the inlist, as shown in Figure 3-26.

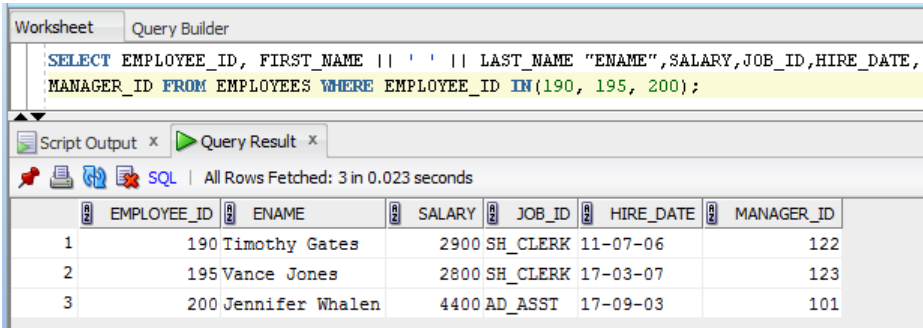
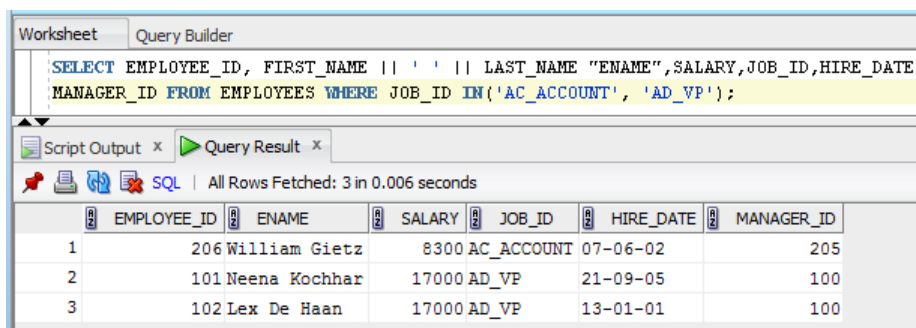


Figure 3-26 Query showing the use of the **IN** operator

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

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
SALARY, JOB_ID, HIRE_DATE, MANAGER_ID FROM EMPLOYEES
WHERE JOB_ID IN('AC_ACCOUNT', 'AD_VP');
```

The above query will list the names of all employees having **AC_ACCOUNT** and **AD_VP** as their **JOB_ID**, as shown in Figure 3-27. In each of these queries, the **IN** operator has been used to select the data based on multiple constant values.



Worksheet Query Builder

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, JOB_ID, HIRE_DATE,
MANAGER_ID FROM EMPLOYEES WHERE JOB_ID IN ('AC_ACCOUNT', 'AD_VP');
```

Script Output x Query Result x

SQL | All Rows Fetched: 3 in 0.006 seconds

EMPLOYEE_ID	ENAME	SALARY	JOB_ID	HIRE_DATE	MANAGER_ID
1	206 William Gietz	8300	AC_ACCOUNT	07-06-02	205
2	101 Neena Kochhar	17000	AD_VP	21-09-05	100
3	102 Lex De Haan	17000	AD_VP	13-01-01	100

Figure 3-27 Query showing the use of the **IN** operator with string inlist

NOT IN

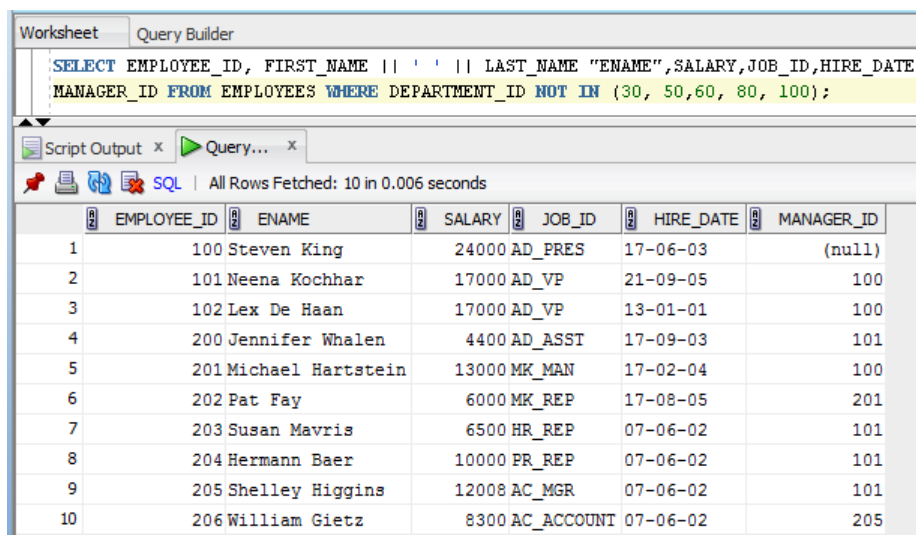
You can combine the **IN** operator with the **NOT** operator. The **NOT IN** operator works just opposite to the **IN** operator. The syntax for using the **NOT IN** operator is as follows:

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

For example:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY,
JOB_ID, HIRE_DATE, MANAGER_ID FROM EMPLOYEES
WHERE DEPARTMENT_ID NOT IN (30, 50, 60, 80, 100);
```

The above query will return the employee details those employees whose department number is not 30, 50, 60, 80, and 100 from the **EMPLOYEES** table. The output of the above query is shown in Figure 3-28.



Worksheet Query Builder

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, JOB_ID, HIRE_DATE,
MANAGER_ID FROM EMPLOYEES WHERE DEPARTMENT_ID NOT IN (30, 50, 60, 80, 100);
```

Script Output x Query... x

SQL | All Rows Fetched: 10 in 0.006 seconds

EMPLOYEE_ID	ENAME	SALARY	JOB_ID	HIRE_DATE	MANAGER_ID
1	100 Steven King	24000	AD_PRES	17-06-03	(null)
2	101 Neena Kochhar	17000	AD_VP	21-09-05	100
3	102 Lex De Haan	17000	AD_VP	13-01-01	100
4	200 Jennifer Whalen	4400	AD_ASST	17-09-03	101
5	201 Michael Hartstein	13000	MK_MAN	17-02-04	100
6	202 Pat Fay	6000	MK_REP	17-08-05	201
7	203 Susan Mavris	6500	HR_REP	07-06-02	101
8	204 Hermann Baer	10000	PR_REP	07-06-02	101
9	205 Shelley Higgins	12008	AC_MGR	07-06-02	101
10	206 William Gietz	8300	AC_ACCOUNT	07-06-02	205

Figure 3-28 Query showing the use of the **NOT IN** operator with string inlist

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 EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, JOB_ID,
HIRE_DATE, DEPARTMENT_ID FROM EMPLOYEES E WHERE EXISTS
(SELECT DEPARTMENT_ID FROM DEPARTMENTS D
WHERE E.DEPARTMENT_ID =60);
```

The output of the above query is shown in Figure 3-29:

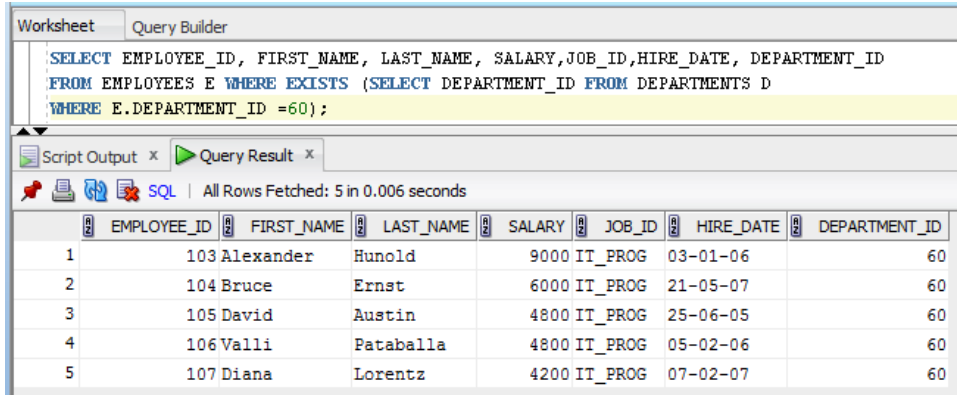


Figure 3-29 Query showing the use of the **EXISTS** operator

NOT EXISTS

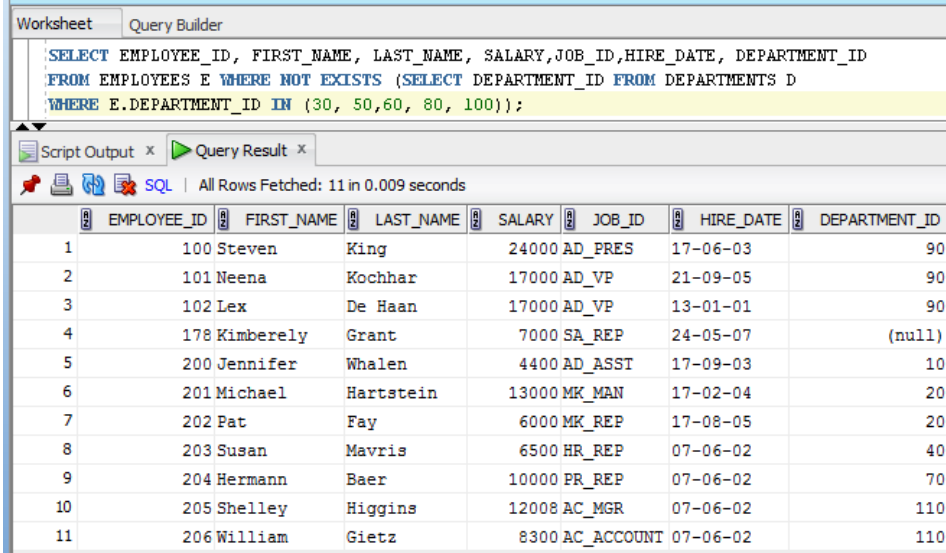
You can also combine the **EXISTS** operator with the **NOT** statement. The **NOT EXISTS** operator works just opposite to the **EXISTS** operator. The syntax for using the **NOT EXISTS** operator is as follows:

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

For example:

```
SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SALARY, JOB_ID,
       HIRE_DATE, DEPARTMENT_ID FROM EMPLOYEES E WHERE NOT EXISTS
       (SELECT DEPARTMENT_ID FROM DEPARTMENTS D
        WHERE E.DEPARTMENT_ID IN (30, 50, 60, 80, 100));
```

The above query will return the number and name of the departments from the **DEPARTMENTS** table, in which there are no records of **DEPARTMENT_ID** in the **EMPLOYEES** table. The output of the above query is shown in Figure 3-30.



	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	JOB_ID	HIRE_DATE	DEPARTMENT_ID
1	100	Steven	King	24000	AD_PRES	17-06-03	90
2	101	Neena	Kochhar	17000	AD_VP	21-09-05	90
3	102	Lex	De Haan	17000	AD_VP	13-01-01	90
4	178	Kimberely	Grant	7000	SA_REP	24-05-07	(null)
5	200	Jennifer	Whalen	4400	AD_ASST	17-09-03	10
6	201	Michael	Hartstein	13000	MK_MAN	17-02-04	20
7	202	Pat	Fay	6000	MK_REP	17-08-05	20
8	203	Susan	Mavris	6500	HR_REP	07-06-02	40
9	204	Hermann	Baer	10000	PR_REP	07-06-02	70
10	205	Shelley	Higgins	12008	AC_MGR	07-06-02	110
11	206	William	Gietz	8300	AC_ACCOUNT	07-06-02	110

Figure 3-30 Query showing the use of the **NOT EXISTS** operator

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 EMPLOYEES WHERE COMMISSION_PCT IS NULL;
```

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

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

```
SELECT * FROM EMPLOYEES WHERE COMMISSION_PCT IS NOT NULL;
```

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

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 JOB_ID FROM EMPLOYEES
UNION ALL
SELECT JOB_ID FROM JOBS;
```

The above example generates a list of job ids from the **EMPLOYEES** and **JOBS** tables.

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 JOB_ID FROM EMPLOYEES
UNION
SELECT JOB_ID FROM JOBS;
```

The above example will generate a list of distinct job ids from the **EMPLOYEES** and **JOBS** tables. 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 EMPLOYEE_ID, JOB_ID, DEPARTMENT_ID FROM EMPLOYEES
UNION
SELECT EMPLOYEE_ID, JOB_ID, DEPARTMENT_ID FROM JOB_HISTORY
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 EMPLOYEE_ID, JOB_ID, DEPARTMENT_ID FROM EMPLOYEES
MINUS
SELECT EMPLOYEE_ID, JOB_ID, DEPARTMENT_ID FROM JOB_HISTORY
ORDER BY 2;
```

The above example will generate a list of employees which are current in job. The above query will return **EMPLOYEE_ID**, **JOB_ID**, and **DEPARTMENT_ID** from the **EMPLOYEES** table which are not in the **JOB_HISTORY** table.

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 DEPARTMENT_ID FROM DEPARTMENTS
INTERSECT
SELECT DEPARTMENT_ID FROM EMPLOYEES;
```

The above example will generate a list of distinct department numbers from the **DEPARTMENTS** and **EMPLOYEES** 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 DEPARTMENT_ID FROM DEPARTMENTS
INTERSECT
SELECT DEPARTMENT_ID FROM EMPLOYEES
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 contains 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.

Table 3-2 The SQL operator 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

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.

CASE EXPRESSION

CASE expression is used to perform multiple condition comparison within a single statement. It is similar to the **IF-THEN-ELSE** statement. It evaluates from top to bottom and if a condition is true, the associated **THEN** clause is executed and the process exits the **CASE** expression by executing the **END** statement. If no condition returns true, it executes the **ELSE** part.

The syntax for using the **CASE** expression is as:

```
CASE [ expression ]
  WHEN condition_1 THEN result_1
  WHEN condition_2 THEN result_2
  ...
  WHEN condition_n THEN result_n
  ELSE result
END
```

The keywords and parameters used in the above syntax are explained next.

expression

It is optional and its value is compared with the list of conditions (ie: **condition_1**, **condition_2**, ... **condition_n**).

condition_1, condition_2, ... condition_n

These conditions are evaluated in the order they are listed and all must be of same data type. Once a condition is evaluated to true, the **CASE** expression returns the result and the rest of the conditions are not evaluated further.

result_1, result_2, ... result_n

These are the values returned once the condition is found to be true. All the results must have the same data type.

For example:

You could use the **CASE** statement in a SQL statement as follows:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME, SALARY,
CASE JOB_ID
  WHEN 'AD_PRES' THEN 'President'
  WHEN 'AC_MGR' THEN 'Accounting Manager'
  WHEN 'AD_VP' THEN 'Administration Vice President'
  WHEN 'FI_MGR' THEN 'Finance Manager'
  ELSE 'ASSISTANT'
END
FROM EMPLOYEES;
```

You can also write the above SQL statement using the **CASE** statement as follows:

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME, SALARY,
CASE
  WHEN JOB_ID='AD_PRES' THEN 'President'
  WHEN JOB_ID='AC_MGR' THEN 'Accounting Manager'
  WHEN JOB_ID='AD_VP' THEN 'Administration Vice President'
  WHEN JOB_ID='FI_MGR' THEN 'Finance Manager'
  ELSE 'ASSISTANT'
END
FROM EMPLOYEES;
```

In the above example, if no condition evaluates to true, then the **CASE** expression will return the value associated with the **ELSE** clause. If the **ELSE** clause is omitted and no condition evaluates to true, the **CASE** expression will return NULL.

SQL CLAUSES

The following are the clauses used in Oracle to retrieve the desired data:

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;
```

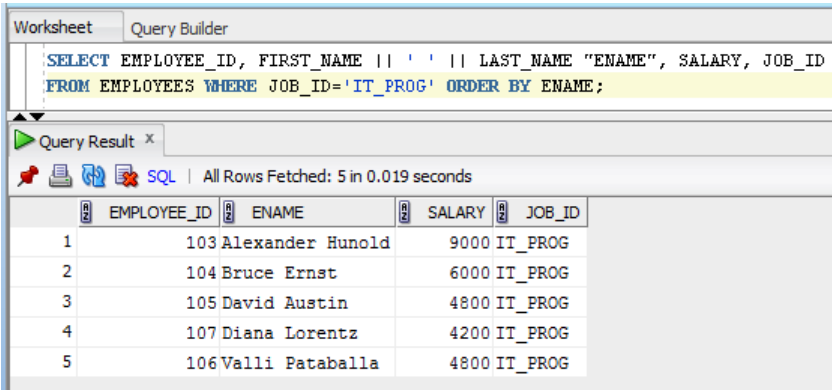
In the above syntax, **ORDER BY** is the keyword and **Column_name** is the name of column of the table **Table_name**. The result will be sorted depending upon the column or columns specified in the **ORDER BY** clause. The keyword **ASC** indicates that the result set will be sorted in the ascending 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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",  
       SALARY, JOB_ID FROM EMPLOYEES  
WHERE JOB_ID='IT_PROG' ORDER BY ENAME;
```

In the above example, the query will return the names of the employees whose **JOB_ID** is **IT_PROG**. 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 **ENAME** field in ascending order, as shown in Figure 3-31.



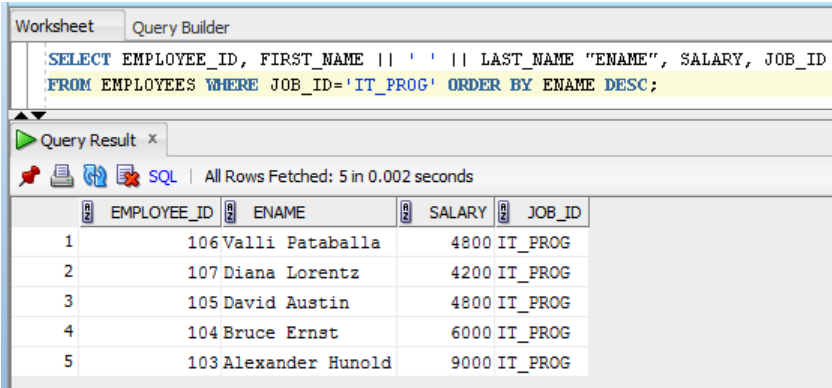
The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying the following SQL query: `SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, JOB_ID FROM EMPLOYEES WHERE JOB_ID='IT_PROG' ORDER BY ENAME;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 5 rows and 4 columns: EMPLOYEE_ID, ENAME, SALARY, and JOB_ID. The records are sorted by ENAME in ascending order.

EMPLOYEE_ID	ENAME	SALARY	JOB_ID
103	Alexander Hunold	9000	IT_PROG
104	Bruce Ernst	6000	IT_PROG
105	David Austin	4800	IT_PROG
107	Diana Lorentz	4200	IT_PROG
106	Valli Pataballa	4800	IT_PROG

Figure 3-31 Sorting records by *ENAME*

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",  
       SALARY, JOB_ID FROM EMPLOYEES  
WHERE JOB_ID='IT_PROG' ORDER BY ENAME DESC;
```

The above query will return the names of employees, whose **JOB_ID** is **IT_PROG**. Here, the records will be sorted by the **ENAME** field in the descending order, as shown in Figure 3-32.



The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying the following SQL query: `SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, JOB_ID FROM EMPLOYEES WHERE JOB_ID='IT_PROG' ORDER BY ENAME DESC;`. Below the query, the 'Query Result' tab shows the results of the query. The results are displayed in a table with 5 rows and 4 columns: EMPLOYEE_ID, ENAME, SALARY, and JOB_ID. The records are sorted by ENAME in descending order.

EMPLOYEE_ID	ENAME	SALARY	JOB_ID
106	Valli Pataballa	4800	IT_PROG
107	Diana Lorentz	4200	IT_PROG
105	David Austin	4800	IT_PROG
104	Bruce Ernst	6000	IT_PROG
103	Alexander Hunold	9000	IT_PROG

Figure 3-32 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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY,
JOB_ID FROM EMPLOYEES WHERE JOB_ID='IT_PROG' ORDER BY 1 DESC;
```

and

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY,
JOB_ID FROM EMPLOYEES WHERE JOB_ID='IT_PROG' ORDER BY 3 DESC;
```

sort the records by the position of fields.

The above queries will return all the records sorted by the position of the field in descending order. The first query will sort the records based on the **EMPLOYEE_ID** field because **EMPLOYEE_ID** is on position 1 in the query. In the second query, the records will be sorted by the **SALARY** field because **SALARY** is on position 3 in the query. The output of the second query is shown in Figure 3-33.

	EMPLOYEE_ID	ENAME	SALARY	JOB_ID
1	103	Alexander Hunold	9000	IT_PROG
2	104	Bruce Ernst	6000	IT_PROG
3	105	David Austin	4800	IT_PROG
4	106	Valli Pataballa	4800	IT_PROG
5	107	Diana Lorentz	4200	IT_PROG

Figure 3-33 Sorting records by the position field

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;
```

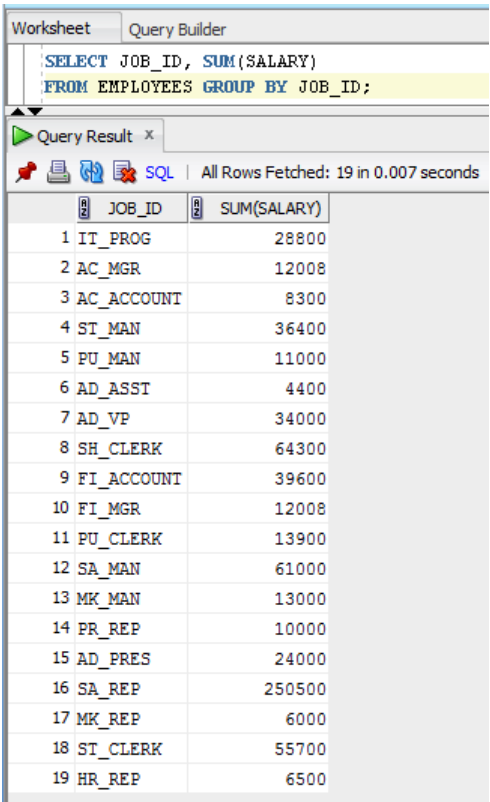
In the above syntax, the **aggregate_Function** can be any aggregate function such as **SUM**, **COUNT**, **MIN**, or **MAX**. These aggregate functions will be discussed in the later chapter.

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_ID, SUM(SALARY) FROM EMPLOYEES GROUP BY JOB_ID;
```

In the above example, the SQL query will return the job id of the employees along with the total salary (for example, total salary of the employees having job id **PU_CLERK**), as shown in Figure 3-34.

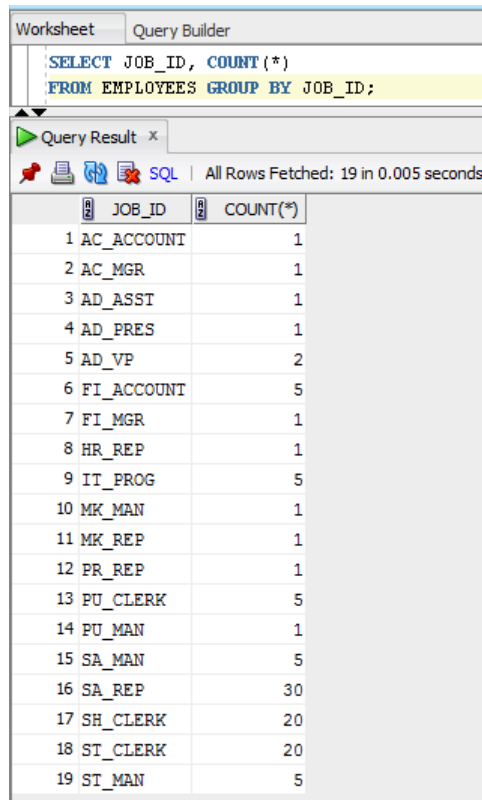


*Figure 3-34 The **GROUP BY** clause with the **SUM** function*

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

```
SELECT JOB_ID, COUNT (*)
FROM EMPLOYEES
GROUP BY JOB_ID;
```

The above SQL query will return the job id of employees with the total number of employees in each job id. Figure 3-35 shows the output of this query.



The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying the following SQL query:

```
SELECT JOB_ID, COUNT (*)
FROM EMPLOYEES GROUP BY JOB_ID;
```

Below the query, the 'Query Result' tab shows the output of the query. It indicates that all 19 rows were fetched in 0.005 seconds. The results are displayed in a table with two columns: JOB_ID and COUNT(*).

JOB_ID	COUNT(*)
AC_ACCOUNT	1
AC_MGR	1
AD_ASST	1
AD PRES	1
AD_VP	2
FI_ACCOUNT	5
FI_MGR	1
HR_REP	1
IT_PROG	5
MK_MAN	1
MK_REP	1
PR_REP	1
PU_CLERK	5
PU_MAN	1
SA_MAN	5
SA_REP	30
SH_CLERK	20
ST_CLERK	20
ST_MAN	5

*Figure 3-35 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_ID, MIN (SALARY)
FROM EMPLOYEES
GROUP BY JOB_ID;
```

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

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


```
SELECT JOB_ID, MAX(SALARY)
FROM EMPLOYEES
GROUP BY JOB_ID;
```

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

For example:

```
SELECT JOB_ID, SUM(SALARY)
FROM EMPLOYEES GROUP BY JOB_ID
HAVING SUM(SALARY) >= 15000;
```

The above SQL query will return the job ids and total salary of the job ids having sum of salaries greater or equal to 15000. The output of the above query is shown in Figure 3-36.

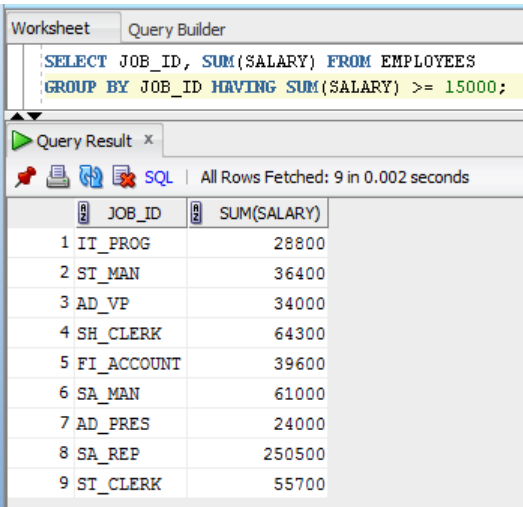
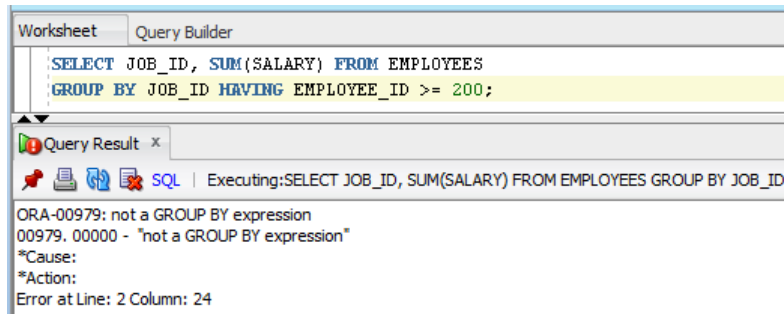


Figure 3-36 Using of the **HAVING** clause

Consider the following query:

```
SELECT JOB_ID, SUM(SALARY)
FROM EMPLOYEES GROUP BY JOB_ID
HAVING EMPLOYEE_ID >= 200;
```

The above query is not valid because **EMPLOYEE_ID** 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 3-37.



*Figure 3-37 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.

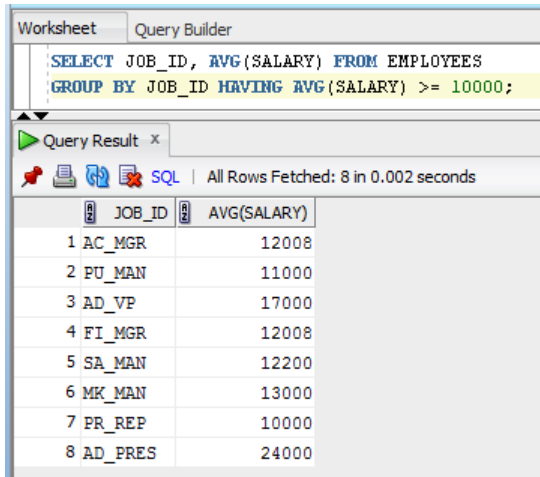
Some examples showing use of the **HAVING** clause with the different aggregate functions are as follows:

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

```
SELECT JOB_ID, AVG(SALARY)
FROM EMPLOYEES
GROUP BY JOB_ID
HAVING AVG(SALARY) >= 10000;
```

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 id's and their average salary which are greater than or equal to 10000, as shown in Figure 3-38.



The screenshot shows a 'Query Builder' window with the following SQL query: `SELECT JOB_ID, AVG(SALARY) FROM EMPLOYEES GROUP BY JOB_ID HAVING AVG(SALARY) >= 10000;`. Below the query, the 'Query Result' section shows 'All Rows Fetched: 8 in 0.002 seconds'. The results are displayed in a table with two columns: 'JOB_ID' and 'AVG(SALARY)'.

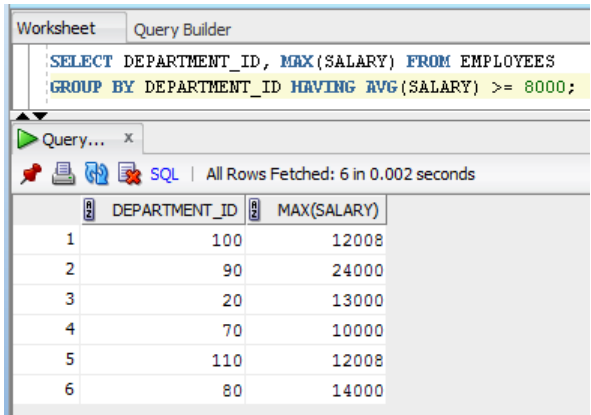
JOB_ID	AVG(SALARY)
1 AC_MGR	12008
2 PU_MAN	11000
3 AD_VP	17000
4 FI_MGR	12008
5 SA_MAN	12200
6 MK_MAN	13000
7 PR_REP	10000
8 AD_PRES	24000

Figure 3-38 Using the *AVG* function

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

```
SELECT DEPARTMENT_ID, MAX(SALARY)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID
HAVING AVG(SALARY) >= 8000;
```

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 the maximum salary of an employee is greater than 8000, as shown in Figure 3-39.



The screenshot shows a 'Query Builder' window with the following SQL query: `SELECT DEPARTMENT_ID, MAX(SALARY) FROM EMPLOYEES GROUP BY DEPARTMENT_ID HAVING AVG(SALARY) >= 8000;`. Below the query, the 'Query Result' section shows 'All Rows Fetched: 6 in 0.002 seconds'. The results are displayed in a table with two columns: 'DEPARTMENT_ID' and 'MAX(SALARY)'.

DEPARTMENT_ID	MAX(SALARY)
100	12008
90	24000
20	13000
70	10000
110	12008
80	14000

Figure 3-39 Using the *HAVING* clause with the *MAX* function

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

```
SELECT DEPARTMENT_ID, MIN(SALARY)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID
HAVING MIN(SALARY) >= 8000;
```

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 8000.

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

```
SELECT DEPARTMENT_ID, SUM(SALARY)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID
HAVING SUM(SALARY) >= 8000;
```

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 8000.

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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY,
JOB_ID, DEPARTMENT_ID FROM EMPLOYEES
WHERE SALARY > (SELECT SALARY FROM EMPLOYEES
                WHERE FIRST_NAME = 'Clara');
```

In the above query, the inner query will return the salary of the employee named **Clara**. 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 shown in Figure 3-40.

The screenshot shows a SQL query in a 'Query Builder' window. The query is: `SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, JOB_ID, DEPARTMENT_ID FROM EMPLOYEES WHERE SALARY > (SELECT SALARY FROM EMPLOYEES WHERE FIRST_NAME = 'Clara');`. Below the query, the 'Query Result' tab shows 13 rows fetched in 0.005 seconds. The results are displayed in a table with columns: EMPLOYEE_ID, ENAME, SALARY, JOB_ID, and DEPARTMENT_ID.

	EMPLOYEE_ID	ENAME	SALARY	JOB_ID	DEPARTMENT_ID
1	100	Steven King	24000	AD_PRES	90
2	101	Neena Kochhar	17000	AD_VP	90
3	102	Lex De Haan	17000	AD_VP	90
4	108	Nancy Greenberg	12008	FI_MGR	100
5	114	Den Raphaely	11000	PU_MAN	30
6	145	John Russell	14000	SA_MAN	80
7	146	Karen Partners	13500	SA_MAN	80
8	147	Alberto Errazuriz	12000	SA_MAN	80
9	148	Gerald Cambrault	11000	SA_MAN	80
10	168	Lisa Ozer	11500	SA_REP	80
11	174	Ellen Abel	11000	SA_REP	80
12	201	Michael Hartstein	13000	MK_MAN	20
13	205	Shelley Higgins	12008	AC_MGR	110

Figure 3-40 Retrieving data using the subquery

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 illustrate 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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY,
JOB_ID, DEPARTMENT_ID FROM EMPLOYEES
WHERE SALARY > (SELECT AVG(SALARY) FROM EMPLOYEES
WHERE JOB_ID=' PU_MAN' );
```

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 shown in Figure 3-41.

Worksheet Query Builder

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME", SALARY, JOB_ID, DEPARTMENT_ID
FROM EMPLOYEES WHERE SALARY > (SELECT AVG(SALARY) FROM EMPLOYEES WHERE JOB_ID='PU_MAN');
```

Query Result x

SQL | All Rows Fetched: 10 in 0.005 seconds

	EMPLOYEE_ID	ENAME	SALARY	JOB_ID	DEPARTMENT_ID
1	100	Steven King	24000	AD_PRES	90
2	101	Neena Kochhar	17000	AD_VP	90
3	102	Lex De Haan	17000	AD_VP	90
4	108	Nancy Greenberg	12008	FI_MGR	100
5	145	John Russell	14000	SA_MAN	80
6	146	Karen Partners	13500	SA_MAN	80
7	147	Alberto Errazuriz	12000	SA_MAN	80
8	168	Lisa Ozer	11500	SA_REP	80
9	201	Michael Hartstein	13000	MK_MAN	20
10	205	Shelley Higgins	12008	AC_MGR	110

Figure 3-41 Retrieving data using the single-row subquery

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

```
SELECT E.*, D.DEPARTMENT_NAME FROM EMPLOYEES E, DEPARTMENTS D
WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID AND
E.SALARY > (SELECT MAX(SALARY) FROM EMPLOYEES
WHERE JOB_ID='PU_MAN');
```

In the above example, the main query will return all the fields from the **EMPLOYEES** table and only one field called the **DEPARTMENT_NAME** from the **DEPARTMENTS** table. The query will return only those records in which the salary of an employee is greater than the maximum salary of the employees having job id **PU_MAN** and have matching values for the **DEPARTMENT_ID** column in both the **EMPLOYEES** and **DEPARTMENT** tables.

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 EMPLOYEES
WHERE EMPLOYEE_ID IN (SELECT EMPLOYEE_ID FROM EMPLOYEES
WHERE COMMISSION_PCT >= 0.25);
```

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 **EMPLOYEES** table whose **COMMISSION_PCT** is greater than or equal to 0.25.

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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",  
       DEPARTMENT_ID FROM EMPLOYEES  
WHERE (EMPLOYEE_ID, DEPARTMENT_ID) IN (SELECT E.EMPLOYEE_ID,  
       D.DEPARTMENT_ID FROM EMPLOYEES E, DEPARTMENTS D  
       WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID);
```

In the above query, the inner query returns two columns, **EMPLOYEE_ID** and **DEPARTMENT_ID**. Here, the comparison is column to column that means the column values are compared as a pair and not individually.

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 EMPLOYEE_ID, MANAGER_ID, FIRST_NAME || ' ' || LAST_NAME  
       "ENAME", SALARY FROM EMPLOYEES OuterE  
WHERE SALARY > (SELECT AVG(SALARY) FROM EMPLOYEES InnerE  
       WHERE InnerE.EMPLOYEE_ID = EMPLOYEE_ID);
```

In the above correlated subquery, you can see that inner query contains a reference to **InnerE.EMPLOYEE_ID**. This reference compares the outer query's **EMPLOYEE_ID** with the inner query's **EMPLOYEE_ID**. 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 whether the inner query returns a value that meets the condition of the **WHERE** clause. The output of the above query is as follows:

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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",
DEPARTMENT_NAME FROM EMPLOYEES, DEPARTMENTS
WHERE EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID;
```

The above SQL query will return the name of the employees with their employee id and department name. It will return only those rows where the department number of the table **EMPLOYEES** matches with department number of the table **DEPARTMENTS**.



Note

*You can add more than one condition to 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

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 EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",  
       DEPARTMENT_NAME FROM EMPLOYEES INNER JOIN DEPARTMENTS  
       ON EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID;
```

The above SQL query will return only those rows, where the department number in the **EMPLOYEES** table matches with the department number in the **DEPARTMENTS** 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 EMPLOYEE_ID, SALARY, DEPARTMENT_NAME  
FROM EMPLOYEES INNER JOIN DEPARTMENTS  
USING (DEPARTMENT_ID);
```

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

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

```
SELECT EMPLOYEE_ID, FIRST_NAME || ' ' || LAST_NAME "ENAME",  
       DEPARTMENT_NAME FROM EMPLOYEES INNER JOIN DEPARTMENTS  
       ON EMPLOYEES.DEPARTMENT_ID = DEPARTMENTS.DEPARTMENT_ID  
       WHERE EMPLOYEES.JOB_ID IN ('AC_MGR', 'ST_MAN', 'IT_PROG');
```

The above SQL query will return only those rows where the department number of the **EMPLOYEES** table matches with the department number of the **DEPARTMENTS** table provided the employee's job is **AC_MGR**, **ST_MAN**, or **IT_PROG**. 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 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 NULL values for the non-matching (that does not follow the join condition) rows of the second table. 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 EMPLOYEES.FIRST_NAME, DEPARTMENTS.DEPARTMENT_NAME
FROM EMPLOYEES LEFT OUTER JOIN DEPARTMENTS
ON EMPLOYEES.DEPARTMENT_ID=DEPARTMENTS.DEPARTMENT_ID;
```

The above SQL query will return all rows from the **DEPARTMENTS** table and only those rows from the **EMPLOYEES** 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 FIRST_NAME, DEPARTMENT_NAME
FROM EMPLOYEES RIGHT OUTER JOIN DEPARTMENTS
ON EMPLOYEES.DEPARTMENT_ID=DEPARTMENTS.DEPARTMENT_ID;
```

The above SQL query will return all rows from the **EMPLOYEES** table and only those rows from the **DEPARTMENTS** 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 EMPLOYEES.FIRST_NAME, EMPLOYEES.SALARY,
DEPARTMENTS.DEPARTMENT_NAME FROM EMPLOYEES FULL OUTER JOIN
DEPARTMENTS ON EMPLOYEES.DEPARTMENT_ID=DEPARTMENTS.DEPARTMENT_ID;
```

The above query will display two columns **FIRST_NAME** and **SALARY** from the table **EMPLOYEES**, and **DEPARTMENT_NAME** from the table **DEPARTMENTS**. It will return only those rows in which the values of the column **DEPARTMENT_ID** of the **EMPLOYEES** table matches with the values of the column **DEPARTMENT_ID** of the **DEPARTMENTS** table. It will also return Null values from both the tables those do 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.FIRST_NAME || ' Is manager of ' || e.FIRST_NAME
FROM EMPLOYEES m, EMPLOYEES e
WHERE m.MANAGER_ID = e.EMPLOYEE_ID;
```

The above query will return both employee number and employee name from the **EMPLOYEES** table, as here the selfjoin retrieves rows from the same table.

Equijoin

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

For example:

```
SELECT e.EMPLOYEE_ID, e.FIRST_NAME, e.SALARY, d.DEPARTMENT_ID,
d.DEPARTMENT_NAME FROM EMPLOYEES e, DEPARTMENTS d
WHERE e.DEPARTMENT_ID = d.DEPARTMENT_ID;
```

The above query will return only those rows in which the department number of the **EMPLOYEES** table matches with the department number of the **DEPARTMENTS** table.

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 EMPLOYEES, DEPARTMENTS;
```

The above query will return each row of the **EMPLOYEES** table with each row of the **DEPARTMENTS** table.

Now consider the following query with the **WHERE** clause:

```
SELECT * FROM EMPLOYEES, DEPARTMENTS
WHERE DEPARTMENTS.DEPARTMENT_ID = 60
AND EMPLOYEES.SALARY > 5000;
```

The above query will return the details of employees of the department 60 having salary greater than 5000 from the **EMPLOYEES** and **DEPARTMENTS** 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 EMPLOYEES
WHERE DEPARTMENT_ID NOT IN
(SELECT DEPARTMENT_ID FROM DEPARTMENTS
WHERE LOCATION_ID = 2500);
```

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 DEPARTMENTS
WHERE EXISTS (SELECT * FROM EMPLOYEES
WHERE EMPLOYEES.DEPARTMENT_ID=DEPARTMENTS.DEPARTMENT_ID)
ORDER BY DEPARTMENT_NAME;
```

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. The output of the above query is shown in Figure 3-42.

The screenshot shows the Oracle Query Builder window. The SQL statement in the top pane is:

```
SELECT * FROM DEPARTMENTS WHERE EXISTS (SELECT * FROM EMPLOYEES
WHERE EMPLOYEES.DEPARTMENT_ID=DEPARTMENTS.DEPARTMENT_ID)
ORDER BY DEPARTMENT_NAME;
```

The bottom pane, titled 'Query Result', shows the results of the query. It indicates 'All Rows Fetched: 11 in 0.003 seconds'. The results are displayed in a table with the following columns: DEPARTMENT_ID, DEPARTMENT_NAME, MANAGER_ID, and LOCATION_ID.

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	110	Accounting	205	1700
2	10	Administration	200	1700
3	90	Executive	100	1700
4	100	Finance	108	1700
5	40	Human Resources	203	2400
6	60	IT	103	1400
7	20	Marketing	201	1800
8	70	Public Relations	204	2700
9	30	Purchasing	114	1700
10	80	Sales	145	2500
11	50	Shipping	121	1500

Figure 3-42 Retrieving data using the Semijoins

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. Oracle 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 **EMPLOYEES** table based on the department number supplied at runtime.

```
SELECT EMPLOYEE_ID, FIRST_NAME, JOB_ID, HIRE_DATE, SALARY
FROM EMPLOYEES
WHERE DEPARTMENT_ID = &DeptID;
```

After executing the above query, the **Enter Substitution Value** input box will be displayed, as shown in Figure 3-43. Enter **30** as the value for the **DeptID** variable. Choose **OK**; the output of the above query will be displayed, as shown in Figure 3-44.

The screenshot shows a dialog box titled 'Enter Substitution Variable'. Inside the dialog, there is a label 'DEPTID:' followed by a text input field. At the bottom of the dialog, there are two buttons: 'OK' and 'Cancel'.

Figure 3-43 The Enter Substitution Variable input box

The screenshot shows a SQL Query Builder window with a 'Worksheet' tab. The query text is: `SELECT EMPLOYEE_ID, FIRST_NAME, JOB_ID, HIRE_DATE, SALARY FROM EMPLOYEES WHERE DEPARTMENT_ID = &DeptID;`. Below the query, there are tabs for 'Script Output' and 'Query Result'. The 'Query Result' tab is active, showing a table with 6 rows and 5 columns: EMPLOYEE_ID, FIRST_NAME, JOB_ID, HIRE_DATE, and SALARY. The data is as follows:

	EMPLOYEE_ID	FIRST_NAME	JOB_ID	HIRE_DATE	SALARY
1	114	Den	PU_MAN	07-12-02	11000
2	115	Alexander	PU_CLERK	18-05-03	3100
3	116	Shelli	PU_CLERK	24-12-05	2900
4	117	Sigal	PU_CLERK	24-07-05	2800
5	118	Guy	PU_CLERK	15-11-06	2600
6	119	Karen	PU_CLERK	10-08-07	2500

Figure 3-44 Retrieving data using Enter Substitution Variable input box

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.

Defining User Variables

You can define substitution variables using the **DEFINE** command to avoid the prompt for the value at runtime.

For example:

To define a user variable named **DeptID** and give it a value **60**, enter and execute the following command:

```
DEFINE DeptID = 60;
```

To confirm the definition of the variable, enter **DEFINE** followed by the variable name:

```
DEFINE DeptID;
```

The following output will be displayed:

```
DEFINE DEPTID = 60 (NUMBER)
```

You can use the above defined variables in the query as input value.

```
SELECT EMPLOYEE_ID, FIRST_NAME, JOB_ID, HIRE_DATE, SALARY,
DEPARTMENT_ID FROM EMPLOYEES WHERE DEPARTMENT_ID = &DeptID;
```

The output of the query is shown in Figure 3-45.

The screenshot shows the Oracle SQL Developer interface. The 'Query Builder' tab is active, displaying a SQL script with a defined variable `DeptID` set to 60. The query selects employee details for department 60. Below the script, the 'Query Result' tab shows a table with 5 rows of data.

	EMPLOYEE_ID	FIRST_NAME	JOB_ID	HIRE_DATE	SALARY	DEPARTMENT_ID
1	103	Alexander	IT_PROG	03-01-06	9000	60
2	104	Bruce	IT_PROG	21-05-07	6000	60
3	105	David	IT_PROG	25-06-05	4800	60
4	106	Valli	IT_PROG	05-02-06	4800	60
5	107	Diana	IT_PROG	07-02-07	4200	60

Figure 3-45 Retrieving data using the defined variables

To delete a user variable, you can use the **UNDEFINE** command followed by the variable name:

```
UNDEFINE DeptID;
```

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**:

```
SELECT &COL1, &COL2, &COL3 FROM &TABLE_NAME
WHERE &COL4 = &VAL;
```

To save the above query, choose the **Save** option from the **File** Menu; the **Save** dialog box will be displayed. In this dialog box, enter the *Demo_Ex* name in the **File name** edit box and choose the **Save** button to save the file. The above query will be saved in the file named *Demo_Ex.sql*. Now, you can execute this file by using the **@** or the **START** command as shown below:

```
START Demo_Ex;
```

After executing the above query, the **Enter Substitution Value** input box will be displayed and you will be prompted to enter value for each variable. The output of the above command is shown in Figure 3-46.

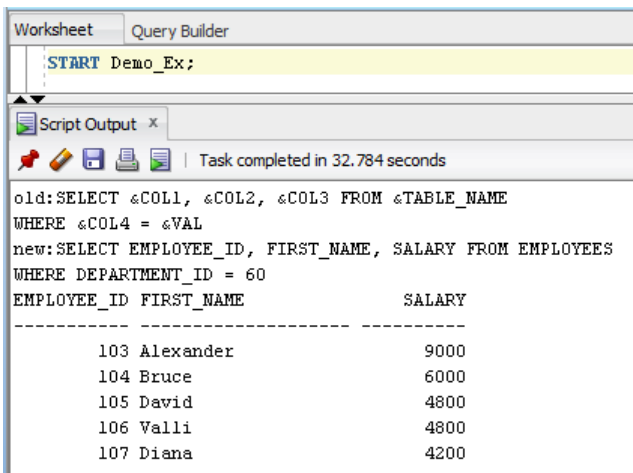


Figure 3-46 Retrieving data using the substitution variables

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:

```
SELECT EMPLOYEE_ID, FIRST_NAME, JOB_ID, HIRE_DATE, SALARY
FROM EMPLOYEES WHERE &1 = &2;
```

After executing the above query, the **Enter Substitution Value** input box will be displayed and you will be prompted to enter values for &1 and &2. Enter **DEPARTMENT_ID** for &1 and **100** for &2. The output of the above command is shown in Figure 3-47.

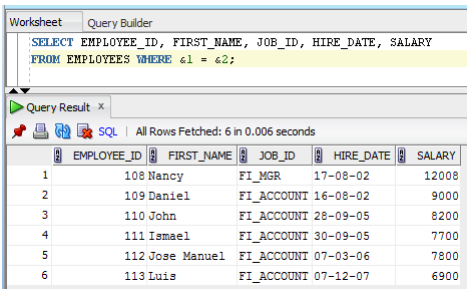


Figure 3-47 Retrieving data using positional notation for variables

PIVOT AND UNPIVOT QUERIES

Pivoting and unpivot are the processes of transposing rows into columns and columns into rows, respectively. In SQL/PLSQL, pivoting and unpivoting can be achieved by using the **PIVOT** and **UNPIVOT** clauses in the **SELECT** statement. These clauses are discussed next:

PIVOT

The **PIVOT** clause is used to transpose rows into columns. It accepts multiple rows, aggregates them and transposes them into columns. The syntax for using the **PIVOT** clause is as follows:

```
SELECT * FROM
(
    SELECT column1, column2
    FROM tables
    WHERE conditions
)
PIVOT
(
    aggregate_function(column2)
    FOR column2
    IN ( expr1, expr2, ... expr_n) | subquery
)
ORDER BY expression [ ASC | DESC ];
```

The keywords and parameters used in the above syntax are explained next.

aggregate_function(column2)

It defines the aggregate function and column(s) to be aggregated.

FOR column2

It defines the columns to be grouped and pivoted.

IN(expr1, expr2, ... expr_n)

It defines the filter for the **column2** to pivot into headings in the cross-tabulation query results.

subquery

It can be used instead of a list of values for **column2**.

For examples:

```
SELECT * FROM
(
    SELECT DEPARTMENT_ID, JOB_ID, SALARY
    FROM EMPLOYEES
)
PIVOT
(
    SUM(SALARY)
    FOR DEPARTMENT_ID
    IN (10, 20, 30, 40, 50, 60)
)
ORDER BY JOB_ID;
```

The above query returns jobs with total salaries in separate columns for each department. The output of the above query is shown in Figure 3-48.

JOB_ID	10	20	30	40	50	60
1 AC_ACCOUNT	(null)	(null)	(null)	(null)	(null)	(null)
2 AC_MGR	(null)	(null)	(null)	(null)	(null)	(null)
3 AD_ASST	4400	(null)	(null)	(null)	(null)	(null)
4 AD_PRES	(null)	(null)	(null)	(null)	(null)	(null)
5 AD_VP	(null)	(null)	(null)	(null)	(null)	(null)
6 FI_ACCOUNT	(null)	(null)	(null)	(null)	(null)	(null)
7 FI_MGR	(null)	(null)	(null)	(null)	(null)	(null)
8 HR_REP	(null)	(null)	(null)	6500	(null)	(null)
9 IT_PROG	(null)	(null)	(null)	(null)	(null)	28800
10 MK_MAN	(null)	13000	(null)	(null)	(null)	(null)
11 MK_REP	(null)	6000	(null)	(null)	(null)	(null)
12 PR_REP	(null)	(null)	(null)	(null)	(null)	(null)
13 PU_CLERK	(null)	(null)	13900	(null)	(null)	(null)
14 PU_MAN	(null)	(null)	11000	(null)	(null)	(null)
15 SA_MAN	(null)	(null)	(null)	(null)	(null)	(null)
16 SA_REP	(null)	(null)	(null)	(null)	(null)	(null)
17 SH_CLERK	(null)	(null)	(null)	(null)	64300	(null)
18 ST_CLERK	(null)	(null)	(null)	(null)	55700	(null)
19 ST_MAN	(null)	(null)	(null)	(null)	36400	(null)

Figure 3-48 Retrieving data using the *PIVOT* clause

You can also alias columns returned by the pivot query:

```

SELECT * FROM
(
    SELECT DEPARTMENT_ID, JOB_ID, SALARY
    FROM EMPLOYEES
)
PIVOT
(
    SUM(SALARY) As Total_Salary
    FOR DEPARTMENT_ID
    IN (10 AS Department_10, 20 AS Department_20, 30 AS Department_30,
        40 AS Department_40, 50 AS Department_50, 60 AS Department_60)
)
ORDER BY JOB_ID;
    
```

The output of the above query is shown in Figure 3-49.

Worksheet Query Builder

```

SELECT * FROM ( SELECT DEPARTMENT_ID, JOB_ID, SALARY FROM EMPLOYEES)
PIVOT ( SUM(SALARY) FOR DEPARTMENT_ID IN (10 AS Department_10, 20 AS Department_20,
30 AS Department_30, 40 AS Department_40, 50 AS Department_50, 60 AS Department_60))
ORDER BY JOB_ID;

```

Query... x All Rows Fetched: 19 in 0.007 seconds

	JOB_ID	DEPARTMENT_10	DEPARTMENT_20	DEPARTMENT_30	DEPARTMENT_40	DEPARTMENT_50	DEPARTMENT_60
1	AC_ACCOUNT	(null)	(null)	(null)	(null)	(null)	(null)
2	AC_MGR	(null)	(null)	(null)	(null)	(null)	(null)
3	AD_ASST	4400	(null)	(null)	(null)	(null)	(null)
4	AD PRES	(null)	(null)	(null)	(null)	(null)	(null)
5	AD_VP	(null)	(null)	(null)	(null)	(null)	(null)
6	FI_ACCOUNT	(null)	(null)	(null)	(null)	(null)	(null)
7	FI_MGR	(null)	(null)	(null)	(null)	(null)	(null)
8	HR_REP	(null)	(null)	(null)	6500	(null)	(null)
9	IT_PROG	(null)	(null)	(null)	(null)	(null)	28800
10	MK_MAN	(null)	13000	(null)	(null)	(null)	(null)
11	MK_REP	(null)	6000	(null)	(null)	(null)	(null)
12	PR_REP	(null)	(null)	(null)	(null)	(null)	(null)
13	PU_CLERK	(null)	(null)	13900	(null)	(null)	(null)
14	PU_MAN	(null)	(null)	11000	(null)	(null)	(null)
15	SA_MAN	(null)	(null)	(null)	(null)	(null)	(null)
16	SA_REP	(null)	(null)	(null)	(null)	(null)	(null)
17	SH_CLERK	(null)	(null)	(null)	(null)	64300	(null)
18	ST_CLERK	(null)	(null)	(null)	(null)	55700	(null)
19	ST_MAN	(null)	(null)	(null)	(null)	36400	(null)

Figure 3-49 Retrieving data using the **PIVOT** clause with column alias

UNPIVOT

The **UNPIVOT** clause is the opposite of the **PIVOT** clause. It is used to transpose columns into rows. The syntax for using the **PIVOT** clause is as follows:

```

SELECT ...
FROM ...
UNPIVOT [INCLUDE|EXCLUDE NULLS]
( unpivot_clause
  unpivot_for_clause
  unpivot_in_clause )
WHERE ...

```

The keywords and parameters used in the above syntax are explained next.

The **INCLUDE | EXCLUDE NULLS** clause gives you the option of including or excluding null-valued rows. If you omit this clause, then the unpivot operation excludes nulls.

unpivot_clause

It specifies a name for each output column that will hold measure values.

unpivot_for_clause

It specifies name for the output column resulting from an unpivot query. The data in this column describes the measure values in the **unpivot_for_clause** column.

unpivot_in_clause

It specifies the input data columns whose names will become values in the output columns of the **unpivot_for_clause**.

For example:

```
WITH emp_data AS (  
    SELECT EMPLOYEE_ID, JOB_ID,  
           FIRST_NAME||' '||LAST_NAME "ENAME",  
           TO_CHAR(DEPARTMENT_ID) AS DEPARTMENT_ID,  
           TO_CHAR(HIRE_DATE) AS HIREDATE  
    FROM EMPLOYEES  
)  
SELECT EMPLOYEE_ID, JOB_ID, unpivot_col_name, unpivot_col_value  
FROM emp_data  
UNPIVOT (unpivot_col_value  
FOR unpivot_col_name  
IN (ENAME, DEPARTMENT_ID, HIREDATE))  
WHERE JOB_ID IN ('AD_PRES', 'AD_VP', 'FI_MGR');
```

The output of the above query is shown in Figure 3-50.

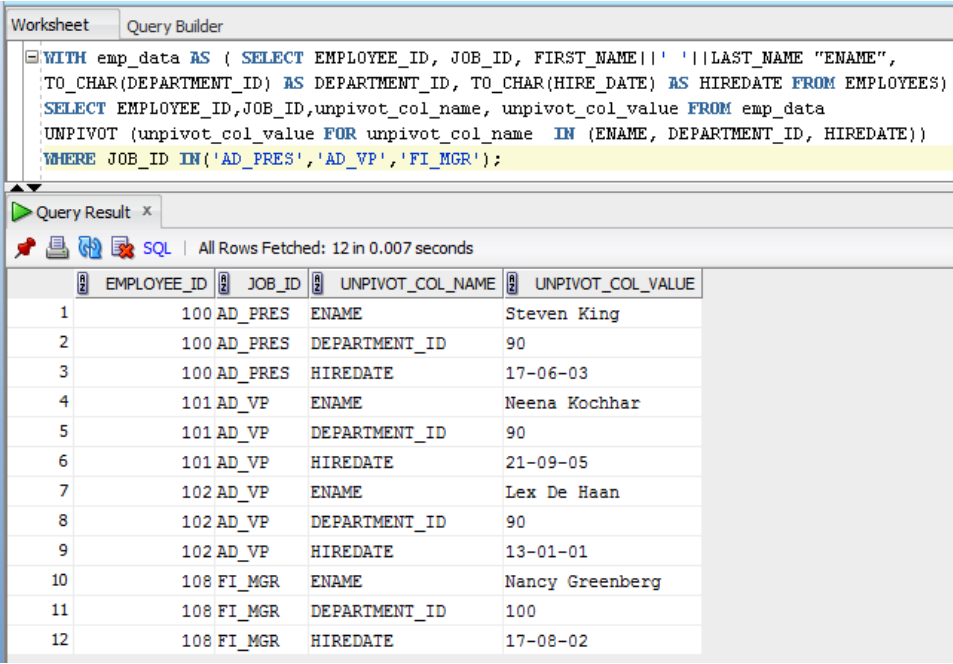


Figure 3-50 Retrieving data using the **UNPIVOT** clause

Self-Evaluation Test

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

1. Which of the following is not a logical operator?
 - (a) **AND**
 - (b) **OR**
 - (c) **NOT**
 - (d) **IN**
2. Which of the following clause is used to filter the data from the database?
 - (a) **WHERE**
 - (b) **DESC**
 - (c) **GROUP BY**
 - (d) **ORDER BY**
3. 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
4. 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**
5. Which of the following is the aggregate function?
 - (a) **SUM**
 - (b) **COUNT**
 - (c) Both (a) and (b)
 - (d) None of these
6. The _____ statement is the most popular SQL statement to query a table.
7. In Oracle, the _____ clause is used to prevent the selection of duplicate rows in a table.
8. The _____ operators are used to compare one expression with another.
9. The _____ clause is used to select, delete, or update only those rows in which the expression evaluates to true.
10. The _____ operator is used to compare the character string with the matching pattern.
11. The **SELECT** statement is used to retrieve or view data from one or more tables. (T/F)
12. You can define the **WHERE** clause with only one condition. (T/F)
13. The **BETWEEN** operator is used to retrieve rows that fall within a specified range. (T/F)

14. The **IN** operator is used to retrieve rows based on the multiple value conditions. (T/F)
15. The **DEFINE** command is used to define a substitution variable in SQL *Plus. (T/F)

Review Questions

Answer the following questions:

- Which of the following operators are used to compare one expression with another expression?
 - Arithmetic
 - Logical
 - Comparison
 - None of these
- Which of the following is the correct syntax for using the **AND** operator?
 - SELECT Column1, Column2.....
From Table
WHERE Condition1 AND Condition2;
 - SELECT Column1, Column2.....
FROM Table
WHERE Condition1 & Condition2:
 - SELECT Column1, Column2.....
FROM Table
WHERE Condition1 && Condition2.
 - None of these
- Which of the following keywords belongs to the SET operators?
 - UNION**
 - MINUS**
 - Both (a) and (b)
 - None of these
- Which of the following operators cannot be applied on the columns of a data type?
 - BLOB**
 - BFILE**
 - Both (a) and (b)
 - None of these
- 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?
 - INNER JOIN**
 - OUTER JOIN**
 - LEFT OUTER JOIN**
 - RIGHT OUTER JOIN**

6. You can define the _____ clause with multiple conditions.
7. The _____ operator joins two or more than two conditions and ensures that the rows satisfying the conditions are selected.
8. The _____ operator joins two or more than two conditions and ensures that the rows satisfying any one of the conditions are selected.
9. The _____ operator joins the result set of two **SELECT** statements.
10. The _____ operator returns distinct rows retrieved by either of the queries.
11. Set operators are used to combine the results from two or more queries into a single result. (T/F)
12. The **UNION** operator returns the difference between two sets. (T/F)
13. Set operators are not used with the **SELECT** statements containing the **TABLE** collection expressions. (T/F)
14. 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)
15. 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)

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