10. INTERACTIVE SQL PART - IV

GROUPING DATA FROM TABLES IN SQL

The Concept Of Grouping

Till now, all SQL **SELECT** statements have:

- Retrieved all the rows from tables
- Retrieved selected rows from tables with the use of a **WHERE** clause, which returns only those rows that meet the conditions specified
- Retrieved unique rows from the table, with the use of **DISTINCT** clause
- Retrieved rows in the sorted order **i.e.** ascending or descending order, as specified, with the use of **ORDER BY** clause.

Other than the above clauses, there are two other clauses, which facilitate selective retrieval of rows. These are the **GROUP BY** and **HAVING** clauses. These are parallel to the **order by** and **where** clause, except that they act on record sets, and **not on** individual records.

GROUP BY Clause

The **GROUP BY** clause is another section of the **select** statement. This optional clause tells Oracle to group rows based on distinct values that exist for **specified columns.** The GROUP BY clause creates a data set, containing several sets of records **grouped together** based on a condition.

Syntax:

```
SELECT <ColumnName1>, <ColumnName2>, <ColumnNameN>,
    AGGREGATE_FUNCTION (<Expression>)
    FROM TableName WHERE <Condition>
        GROUP BY <ColumnName1>, <ColumnName2>, <ColumnNameN>;
```

Example 1:
Find out how many employees are there in each branch.

Synopsis:

Tables:	EMP_MSTR
Columns:	BRANCH_NO, EMP_NO
Technique:	**Functions:** COUNT(), **Clauses:** GROUP BY, **Others:** Alias

Solution:

```
SELECT BRANCH_NO "Branch No.", COUNT(EMP_NO) "No. Of Employees"
    FROM EMP_MSTR GROUP BY BRANCH_NO;
```

Output:

```
Branch No.  No. Of Employees
----------  ----------------
B1                         2
B2                         2
B3                         2
B4                         2
B6                         2
```

Explanation:
In the above example, the data that has to be retrieved is available in the **EMP_MSTR** table. Since the **number of employees** per branch is required, the records need to be **grouped** on the basis of field **BRANCH_NO** and then the **COUNT()** function must be applied to the field **EMP_NO** which calculates the number of employees on a per branch basis.

Example 2:
Find out the total number of (Current and Savings Bank) accounts verified by each employee.

Synopsis:

Tables:	ACCT_MSTR
Columns:	VERI_EMP_NO, ACCT_NO
Technique:	**Functions:** COUNT(), **Clauses:** GROUP BY, **Others:** Alias

Solution:
```
SELECT VERI_EMP_NO "Emp. No.", COUNT(ACCT_NO) "No. Of A/Cs Verified"
   FROM ACCT_MSTR GROUP BY VERI_EMP_NO;
```

Output:
```
Emp. No.   No. Of A/Cs Verified
E1                            7
E4                            8
```

Explanation:
In the above example, the data that has to be retrieved is available in the **ACCT_MSTR** table. Since the **number of accounts verified** per employee is required, the records need to be **grouped** on the basis of field **VERI_EMP_NO** and then the **COUNT()** function is applied to the field **ACCT_NO** which calculates the number of accounts verified per employee.

Example 3:
Find out the total number of accounts segregated on the basis of account type per branch.

Synopsis:

Tables:	ACCT_MSTR
Columns:	BRANCH_NO, TYPE, ACCT_NO
Technique:	**Functions:** COUNT(), **Clauses:** GROUP BY, **Others:** Alias

Solution:
```
SELECT BRANCH_NO "Branch No.", TYPE "A/C Type", COUNT(ACCT_NO) "No. Of A/Cs"
   FROM ACCT_MSTR GROUP BY BRANCH_NO, TYPE;
```

Output:
```
Branch No.  A/C Type  No. Of A/Cs
B1          CA                  1
B1          SB                  2
B2          CA                  2
B2          SB                  1
B3          SB                  2
B4          SB                  2
B5          SB                  2
B6          CA                  1
B6          SB                  2
9 rows selected.
```

Explanation:
In the above example, the data that has to be retrieved is available in the **ACCT_MSTR** table. Since the **number of accounts** based on the account type per branch is required, the records need to be **grouped** on the basis of two fields i.e. **BRANCH_NO** and within it **TYPE** and then the **COUNT()** function is applied to the field **ACCT_NO** which calculates the number of accounts of a particular type in a particular branch.

HAVING Clause

The **HAVING** clause can be used in conjunction with the **GROUP BY** clause. **HAVING** imposes a condition on the **GROUP BY** clause, which further filters the groups created by the **GROUP BY** clause. Each column specification specified in the HAVING clause must occur within a statistical function or must occur in the list of columns named in the GROUP BY clause.

Example 4:
Find out the customers having more than one account in the bank.

Synopsis:

Tables:	ACCT_FD_CUST_DTLS
Columns:	CUST_NO, ACCT_FD_NO
Technique:	**Functions:** COUNT(), **Operators:** LIKE, OR, **Clauses:** GROUP BY ... HAVING, **Others:** Alias

Solution:
```
SELECT CUST_NO, COUNT(ACCT_FD_NO) "No. Of A/Cs Held" FROM ACCT_FD_CUST_DTLS
    WHERE ACCT_FD_NO LIKE 'CA%' OR ACCT_FD_NO LIKE 'SB%'
        GROUP BY CUST_NO HAVING COUNT(ACCT_FD_NO)>1;
```

Output:
```
CUST_NO  No. Of A/Cs Held
-------  ----------------
C1                      4
C10                     2
C2                      2
C3                      3
C4                      6
C5                      3
C9                      2
7 rows selected.
```

Explanation:
In the above example, the data that has to be retrieved is available in the Accounts-F.D.-Customers link table (i.e. **ACCT_FD_CUST_DTLS**). This table holds data related to accounts as well as fixed deposits. Since, only the data related to accounts is required there is a need to **filter** the data. This is done using the LIKE operator, which will only retrieve the records related to Current and Savings Bank Accounts (i.e. value held in the **ACCT_FD_NO** field beginning with **CA** or **SB**). The **Count()** function is applied to the field **ACCT_FD_NO** which will now hold only the filtered values i.e. either **CA__** or **SB__**. This filtered information is then **grouped** on the basis of Customer Number (i.e. the **CUST_NO** field). Since only those customers who hold more than one account are to be retrieved, the **HAVING** clause is used to finally filter the data to retain only those records where the value calculated using the **COUNT()** function is **greater than 1**.

Example 5:
Find out the number of accounts opened at a branch after **03rd January 2003**, only if the number of accounts opened after 03rd January 2003 **exceeds 1**.

Synopsis:

Tables:	ACCT_MSTR
Columns:	BRANCH_NO, ACCT_NO
Technique:	**Functions:** COUNT(),TO_CHAR(), **Operators:** LIKE, **Clauses:** GROUP BY ... HAVING, **Others:** Alias

Solution:
```
SELECT BRANCH_NO, COUNT(ACCT_NO) "No. Of A/Cs Activated"
    FROM ACCT_MSTR WHERE TO_CHAR(OPNDT, 'DD-MM-YYYY') > '03-01-2003'
        GROUP BY BRANCH_NO HAVING COUNT(ACCT_NO) > 1;
```

Output:
```
BRANCH_NO  No. Of A/Cs Activated
---------- ---------------------
B1                             3
B2                             3
B3                             2
B4                             2
B5                             2
B6                             3
6 rows selected.
```

Explanation:
In the above example, the data that has to be retrieved is available in the ACCT_MSTR table. This table holds data related to all the accounts that have been activated under a particular branch. Since, only the data related to a particular date is required there is a need to **filter** the data. This is done using the **WHERE** clause and the **TO_CHAR()** function which converts the date to character format and makes it available for comparison with the date **'3rd January 2003'**, which will only retrieve the accounts opened after that date. (i.e. value held in the **OPNDT** field is greater than **'3rd January 2003'**). The **Count()** function is applied to the field **ACCT_NO** which will now hold only the filtered values i.e. those accounts which were activated after **'3rd January 2003'**. This filtered information is then **grouped** on the basis of Branch Number (i.e. the **BRANCH_NO** field). Since only those records that have the count of account numbers **more than 1** are to be retrieved, the **HAVING** clause is used to finally filter the data to retain only those records where the value calculated using the **COUNT()** function is **greater than 1**.

Rules For Group By and Having Clause
- Columns listed in the select statement have to be listed in the GROUP BY clause
- Columns listed in the GROUP BY clause need not be listed in the SELECT statement
- Only group functions can be used in the HAVING clause
- The group functions listed in the having clause need not be listed in the SELECT statement

Determining Whether Values Are Unique

The **HAVING** clause can be used to find unique values in situations to which DISTINCT does not apply.

The **DISTINCT** clause eliminates duplicates, but does not show which values actually were duplicated in the original data. The **HAVING** clause can identify which values were unique or non-unique.

Example 6:
List customer numbers, which are associated with only one account (or Fixed deposit) in the bank. (Unique Entries Only)

Synopsis:

Tables:	ACCT_FD_CUST_DTLS
Columns:	CUST_NO, ACCT_FD_NO
Technique:	**Functions:** COUNT(), **Clauses:** GROUP BY ... HAVING, **Others:** Alias

Solution:
```
SELECT CUST_NO, COUNT(ACCT_FD_NO) "No. Of A/Cs Or FDs Held"
    FROM ACCT_FD_CUST_DTLS GROUP BY CUST_NO HAVING COUNT(ACCT_FD_NO) = 1;
```

Output:
```
CUST_NO  No. Of A/Cs Or FDs Held
-------- -------------------------
C7                              1
```

Explanation:
In the above example, the data that has to be retrieved is available in the Accounts-F.D.-Customers link table (i.e. **ACCT_FD_CUST_DTLS**). This table holds data related to accounts as wells as fixed deposits. The **Count()** function is applied to the field **ACCT_FD_NO** which will hold the number of accounts or fixed deposits held by a particular customer. This information is then **grouped** on the basis of Customer Number (i.e. the **CUST_NO** field). Since only those customers who hold **only one** account or fixed deposit are to be retrieved, the **HAVING** clause is used to finally filter the data to retain only those records where the value calculated using the **COUNT()** function is **equal to 1**.

Example 7:
List the customer numbers associated with **more** than one account (or Fixed deposits). (Non-Unique Entries)

Synopsis:

Tables:	ACCT_FD_CUST_DTLS
Columns:	CUST_NO, ACCT_FD_NO
Technique:	**Functions:** COUNT(), **Clauses:** GROUP BY ... HAVING, **Others:** Alias

Solution:
```
SELECT CUST_NO, COUNT(ACCT_FD_NO) "No. Of A/Cs or FDs Held"
    FROM ACCT_FD_CUST_DTLS GROUP BY CUST_NO HAVING COUNT(ACCT_FD_NO)>1;
```

Output:
```
CUST_NO  No. Of A/Cs Or FDs Held
-------- -------------------------
C1                              4
C10                             3
C2                              3
C3                              4
C4                              7
C5                              6
C6                              2
C8                              2
C9                              3
9 rows selected.
```

Explanation:
In the above example, the data that has to be retrieved is available in the Accounts-F.D.-Customers link table (i.e. **ACCT_FD_CUST_DTLS**).

This table holds data related to accounts as wells as fixed deposits. The **COUNT()** function is applied to the field **ACCT_FD_NO** which will hold the number of accounts or fixed deposits held by a particular customer. This information is then **grouped** on the basis of Customer Number (i.e. the **CUST_NO** field). Since only those customers who hold more than one account or fixed deposit are to be retrieved, the **HAVING** clause is used to finally filter the data to retain only those records where the value calculated using the **COUNT()** function is **greater than 1**.

Group By Using The ROLLUP Operator

The ROLLUP operator is used to calculate aggregates and super aggregates for expressions within a GROUP BY statement. Report writers usually use this operator to extract statistics and/or summaries from a result set.

Example 8:
Create a report on the fixed deposits accounts available in the bank, providing the **amount** and the **due amount** per fixed deposit (per FD_NO in the FD_DTLS table) and per slot (per FD_SER_NO in the FD_MSTR table) of fixed deposit held by the customer.

Synopsis:

Tables:	FD_MSTR
Columns:	FD_SER_NO, FD_NO, AMT, DUEAMT
Technique:	**Functions:** SUM(), **Operators:** ROLLUP(), **Clauses:** GROUP BY

Solution:

```
SELECT FD_SER_NO, FD_NO, SUM(AMT), SUM(DUEAMT)
   FROM FD_DTLS GROUP BY ROLLUP (FD_SER_NO, FD_NO);
```

Output:

FD_SER_NO	FD_NO	AMT	DUEAMT
FS1	F1	15000	16050
FS1	F2	5000	5350
FS1		**20000**	**21400**
FS2	F3	10000	10802.19
FS2	F4	10000	10802.19
FS2		**20000**	**21604.38**
FS3	F5	2000	2060.16
FS3		**2000**	**2060.16**
FS4	F6	5000	5902.47
FS4		**5000**	**5902.47**
FS5	F7	15000	16203.3
FS5		**15000**	**16203.3**
		62000	***67170.31***

13 rows selected.

Explanation:
In the above example, the data that has to be retrieved is available in the **FD_DTLS** table. This table holds data related to individual fixed deposits (i.e. per FD_NO) **held within a fixed deposit slot (i.e. per FD_SER_NO in the FD_MSTR table).** The **SUM()** function is applied on the field **AMT** and **DUEAMT** which will hold the sum of amount and due amount per FD with a slot. This information is then grouped on the basis of **FD_SER_NO** and **FD_NO** using the **ROLLUP** operator.

The **ROLLUP** operator is used to display the amount and due amount per fixed deposit (i.e. per FD_NO) and the total amount and total due amount per fixed deposit slot (i.e. per FD_SER_NO).

The ROLLUP operator first calculates the standard aggregate values for the groups specified in the group by clause (**SUM** of **AMT** and **DUEAMT** for each fixed deposit i.e. per **FD_NO**) then creates higher level subtotals, moving from right to left through the list of grouping columns (**SUM** of **AMT** and **DUEAMT** for each fixed deposit slot i.e. per **FD_SER_NO**).

Grouping By Using The CUBE Operator

The **CUBE** operator can be applied to all aggregates functions like **AVG()**, **SUM()**, **MAX()**, **MIN()** and **COUNT()** within a **GROUP BY** statement. This operator is usually used by, report writers to extract cross-tabular reports from a result set. CUBE produces subtotals for all possible combinations of grouping specified in the **GROUP BY** clause along with a grand total as against the ROLLUP operator which produces only a fraction of possible subtotal combinations.

Example 9:
Find out the Balance of the account holders on per account and per branch basis along with a grand total.

Synopsis:

Tables:	ACCT_MSTR
Columns:	BRANCH_NO, ACCT_NO, CURBAL
Technique:	**Functions:** SUM(), **Operators:** CUBE(), **Clauses:** GROUP BY

Solution:
```
SELECT BRANCH_NO, ACCT_NO, SUM(CURBAL) FROM ACCT_MSTR
      GROUP BY CUBE (BRANCH_NO, ACCT_NO);
```

Output:

BRANCH_NO	ACCT_NO	SUM(CURBAL)
		88500
	CA2	3000
	CA4	12000
	CA7	22000
	SB1	500
	SB3	500
	SB5	500
	SB6	500
	SB8	500
	SB9	500
	CA10	32000
	CA12	5000
	CA14	10000
	SB11	500
	SB13	500

Output: (Continued)

BRANCH_NO	ACCT_NO
	SB15
	CA7
B1	SB1
B1	SB11
B1	
B1	CA2
B2	SB8
B2	CA1
B2	
B2	SB
B3	SB
B3	
B3	S
B4	S
B4	
B4	
B5	C
B5	C
B5	
B6	
B6	
B6	
B6	

37 rows sel

Explanation:
In the above ex
holds data relat
field **CURBAL**
basis of **BRAN**

The balance o
is used to dis
balance per
irrespective o

The **CUBE**
clause (Sum
through the
displays the
irrespective

SUBQU

A **subque**
nested qu
the rows

Output: (Continued)

```
BRANCH_NO  ACCT_NO  SUM(CURBAL)
----------------------------------
           SB15              500
                           23000
B1         CA7             22000
B1         SB1               500
B1         SB11              500
B1                          8500
B2         CA2              3000
B2         SB8               500
B2         CA12             5000
B2                          1000
B3         SB3               500
B3         SB13              500
B3                          1000
B4         SB6               500
B4         SB9               500
B4                         22000
B5         CA4              12000
B5         CA14             10000
B5                         33000
B6         SB5               500
B6         CA10             32000
B6         SB15              500
B6
37 rows selected.
```

Explanation:

In the above example, the data that has to be retrieved is available in the **ACCT_MSTR** table. This table holds data related to savings and current accounts held by the bank. The **SUM()** function is applied to the field **CURBAL** which will hold the sum of balance per account. This information is then **grouped** on the basis of **BRANCH_NO** and **ACCT_NO** using the **CUBE** operator.

The balance of every account within a branch is displayed using the group by clause. The **CUBE** operator is used to display the **balance per account**, the **total balance** held in **each branch** of the bank, the total balance per account irrespective of the branch and the total balance held in all branches of the bank irrespective of the accounts held.

The **CUBE** operator first calculates the standard aggregate values for the groups specified in the group by clause (Sum of **CURBAL** for each account) then creates higher level subtotals, moving from right to left through the list of grouping columns (Sum of **CURBAL** for each branch). Additionally the **CUBE** operator displays the total balance per account irrespective of branch and the total balance of all the branches irrespective of the accounts held.

SUBQUERIES

A **subquery** is a form of an SQL statement that appears inside another SQL statement. It is also termed as **nested query**. The statement containing a subquery is called a **parent** statement. The parent statement uses the rows (i.e. the result set) returned by the subquery.

It can be used for the following:

- To insert records in a target table
- To create tables and insert records in the table created
- To update records in a target table
- To create views
- To provide values for **conditions** in WHERE, HAVING, IN and so on used with SELECT, UPDATE and DELETE statements

Example 10:
Retrieve the address of a customer named 'Ivan Bayross'.

Synopsis:

Tables:	CUST_MSTR, ADDR_DTLS
Columns:	CUST_MSTR: CUST_NO, FNAME, LNAME ADDR_DTLS: CODE_NO, ADDR1, ADDR2, CITY, STATE, PINCODE
Technique:	**Sub-Queries, Operators:** IN, **Clauses:** WHERE, **Other:** Concat (||)

Solution:

```
SELECT CODE_NO "Cust. No.", ADDR1 || ' ' || ADDR2 || ' ' || CITY || ', ' || STATE || ', ' || PINCODE
        "Address"
    FROM ADDR_DTLS WHERE CODE_NO IN(SELECT CUST_NO FROM CUST_MSTR
        WHERE FNAME = 'Ivan' AND LNAME = 'Bayross');
```

Output:

```
Cust. No.  Address
-----------------------------------------------------------------------------
C1         F-12, Diamond Palace, West Avenue, North Avenue, Santacruz
           (West), Mumbai, Maharashtra, 400056
```

Explanation:
In the above example, the data that has to be retrieved is available in the **ADDR_DTLS** table, which holds the address for customer named **'Ivan Bayross'**. This table holds all the address details identified by the customer number i.e. **CODE_NO**. However, the **ADDR_DTLS** table does not contain the field, which holds the customer's name, which is required to make a comparison.

The Customers Name is available in the **CUST_MSTR** table where each customer is identified by a unique number (i.e. **CUST_NO**). So it is required to access the table **CUST_MSTR** and retrieve the **CUST_NO** based on which a comparison can be made with the **CODE_NO** field held in the table **ADDR_DTLS**.

Using the **CUST_NO** retrieved from the **CUST_MSTR** table, it is now possible to retrieve the address(es) from the **ADDR_DTLS** table by finding a matching value in the **CODE_NO** field in that table.

This type of processing can be done elegantly using a subquery.

In the above solution the sub-query is as follows:

```
SELECT CUST_NO FROM CUST_MSTR
    WHERE FNAME = 'IVAN' AND LNAME = 'BAYROSS';
```

The target table will be as follows:

Output:

```
CUST_NO
-------
C1
```

The outer sub-query output will simplify the solution as shown below:

```
SELECT CODE_NO "Cust. No.", ADDR1 || ' ' || ADDR2 || ' ' || CITY || ', ' || STATE || ', ' || PINCODE
        "Address" FROM ADDR_DTLS WHERE CODE_NO IN('C1');
```

When the above SQL query is executed the resulting output is equivalent to the desired output of the SQL query using two levels of Sub-queries.

Example 11:

Find the customers who do not have bank branches in their vicinity.

Synopsis:

Tables:	CUST_MSTR, ADDR_DTLS
Columns:	CUST_MSTR: CUST_NO, FNAME, LNAME ADDR_DTLS: CODE_NO, PINCODE
Technique:	**Sub-Queries, Operators:** IN, **Clauses:** WHERE, **Other:** Concat (||)

Solution:

```
SELECT (FNAME || ' ' || LNAME) "Customer" FROM CUST_MSTR
    WHERE CUST_NO IN(SELECT CODE_NO FROM ADDR_DTLS
        WHERE CODE_NO LIKE 'C%' AND PINCODE NOT IN(SELECT PINCODE
        FROM ADDR_DTLS WHERE CODE_NO LIKE 'B%'));
```

Output:

```
Customer
--------------------------------
Ivan Bayross
Namita Kanade
Chriselle Bayross
Mamta Muzumdar
Chhaya Bankar
Ashwini Joshi
Hansel Colaco
Anil Dhone
Alex Fernandes
Ashwini Apte
10 rows selected.
```

Explanation:

In the above example, the data that has to be retrieved is available in the **CUST_MSTR**, which holds the Customer details. The **CUST_MSTR** table will only provide all the customer names but to retrieve only those customers who do not have any bank branches in their vicinity, one more tables will be involved that is **ADDR_DTLS** table, which holds the branch as well as customer addresses. This table holds all the address details identified by the branch / customer number i.e. **CODE_NO**. This table will help comparing the value held in the **PINCODE** field belonging to the customer addresses with the ones of bank branch addresses.

To understand the solution the query mentioned above needs to be simplified. The inner most sub-queries should be handled first and then proceeded outwards.

The **first step** is to identify the vicinities in which the branches are located. This is done by extracting the **PINCODE** from the address details table (i.e. **ADDR_DTLS**) for all entries belonging to the branches. The SQL query for this will be as follows:

```
SELECT PINCODE FROM ADDR_DTLS WHERE CODE_NO LIKE 'B%'
```

The target table will be as follows:

Output:

```
PINCODE
--------------
400057
400058
400004
400045
400078
110004
6 rows selected.
```

The resulting output simplifies the solution as shown below:

```
SELECT (FNAME || ' ' || LNAME) "Customer" FROM CUST_MSTR
    WHERE CUST_NO IN(SELECT CODE_NO FROM ADDR_DTLS
        WHERE CODE_NO LIKE 'C%' AND PINCODE NOT IN('400057', '400058',
                                                   '400004', '400045', '400078', '110004'));
```

The **second step** is to identify the customers who are not resident near a branch. To do this the customer numbers (i.e. **CODE_NO**) have to be retrieved from the Address details table (i.e. **ADDR_DTLS**). The SQL query for this will be as follows:

```
SELECT CODE_NO FROM ADDR_DTLS WHERE CODE_NO LIKE 'C%'
    AND PINCODE NOT IN('400057', '400058', '400004', '400045', '400078', '110004')
```

The target table will be as follows:

Output:

```
CODE_NO
--------------
C1
C2
C3
C4
C5
C6
C7
C8
C9
C10
10 rows selected.
```

The outer sub-query output will simplify the solution as shown below:

```
SELECT (FName || ' ' || LName) "Customer" FROM Cust_Mstr
    WHERE Cust_No IN('C1', 'C2', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9', 'C10');
```

When the above SQL query is executed the resulting output is equivalent to the desired output of the SQL query using two levels of Sub-queries.

Example 12:

List customers holding Fixed Deposits in the bank of amount more than 5,000.

Synopsis:

Tables:	CUST_MSTR, ACCT_FD_CUST_DTLS, FD_DTLS
Columns:	CUST_MSTR: CUST_NO, FNAME, LNAME ACCT_FD_CUST_DTLS: CUST_NO, ACCT_FD_NO FD_DTLS: FD_SER_NO, AMT
Technique:	**Sub-Queries, Operators:** IN() **Clauses:** WHERE, **Other:** Concat (||)

Solution:

```
SELECT (FNAME || ' ' || LNAME) "Customer" FROM CUST_MSTR WHERE CUST_NO
   IN(SELECT CUST_NO FROM ACCT_FD_CUST_DTLS WHERE ACCT_FD_NO
      IN(SELECT FD_SER_NO FROM FD_DTLS WHERE AMT > 5000));
```

Output:

```
Customer
--------------------
Chriselle Bayross
Mamta Muzumdar
Chhaya Bankar
Ashwini Joshi
```

Explanation:

In the above example, the data that has to be retrieved is available in the **CUST_MSTR**, which holds the Customer details. The **CUST_MSTR** table will only provide all the customer names but to retrieve only those customers who hold fixed deposits of amount exceeding Rs.5000, two more tables will be involved that is **ACCT_FD_DTLS** and **FD_DTLS.** The **ACCT_FD_DTLS** acts as a link between the **CUST_MSTR** and **FD_DTLS** table and hold details identified by the **ACCT_FD_NO** field. The **FD_DTLS** table actually hold all the details related to the fixed deposits held by the customers in a bank identified by the **FD_SER_NO** field.

To understand the solution the query mentioned above needs to be simplified. The inner most sub-queries should be handled first and then proceeded outwards.

The **first step** is to identify the fixed deposits of amount more than 5000. This is done by extracting the value held by the **FD_SER_NO** field from the fixed deposits details table (i.e. **FD_DTLS**). The SQL query for this will be as follows:

```
SELECT FD_SER_NO FROM FD_DTLS WHERE AMT > 5000
```

The target table will be as follows:

Output:

```
FD_SER_NO
---------
FS1
FS2
FS2
FS5
```

The data retrieved by the above **SELECT** statement will be passed to the outer sub-query as in:

```
SELECT (FNAME || ' ' || LNAME) "Customer" FROM CUST_MSTR WHERE CUST_NO
   IN(SELECT CUST_NO FROM ACCT_FD_CUST_DTLS WHERE ACCT_FD_NO
      IN('FS1', 'FS2', 'FS2', 'FS5'));
```

The **second step** is to identify the customer numbers who hold these FDs i.e. ('FS1', 'FS2', 'FS2', 'FS5'). To do this the customer numbers (i.e. **CUST_NO**) have to be retrieved from the Account FD details table (i.e. **ACCT_FD_DTLS**). The SQL query for this will be as follows:

```
SELECT CUST_NO FROM ACCT_FD_CUST_DTLS
   WHERE ACCT_FD_NO IN('FS1', 'FS2', 'FS2', 'FS5')
```

The target table will be as follows:

Output:

```
CUST_NO
-------
C2
C3
C4
C5
C5
C5
6 rows selected.
```

The outer sub-query output will simplify the solution as shown below:

```
SELECT (FNAME || ' ' || LNAME) "Customer" FROM CUST_MSTR
    WHERE CUST_NO IN('C2', 'C3', 'C4', 'C5', 'C5', 'C5');
```

When the above SQL query is executed the resulting output is equivalent to the desired output of the SQL query using two levels of Sub-queries.

Using Sub-query In The FROM Clause

A subquery can be used in the **FROM** clause of the **SELECT** statement. The concept of using a subquery in the **FROM** clause of the **SELECT** statement is called an **inline view**. A subquery in the **FROM** clause of the **SELECT** statement defines a **data source** from that particular **Select** statement.

Example 13:
List accounts along with the current balance, the branch to which it belongs and the average balance of the branch, having a balance more than the average balance of the branch, to which the account belongs.

Synopsis:

Tables:	ACCT_MSTR
Columns:	ACCT_NO, CURBAL, BRANCH_NO
Technique:	**Sub-Queries, Join, Functions:** AVG(), **Clauses:** WHERE, GROUP BY

Solution:

```
SELECT A.ACCT_NO, A.CURBAL, A.BRANCH_NO, B.AVGBAL
    FROM ACCT_MSTR A, (SELECT BRANCH_NO, AVG(CURBAL) AVGBAL FROM ACCT_MSTR
        GROUP BY BRANCH_NO) B
    WHERE A.BRANCH_NO = B.BRANCH_NO AND A.CURBAL > B.AVGBAL;
```

Output:

```
ACCT_NO   CURBAL BRANCH_NO   AVGBAL
-------  -------  ---------  -------
CA7        22000  B1         7666.67
CA2         3000  B2         2833.33
CA12        5000  B2         2833.33
CA4        12000  B5           11000
CA10       32000  B6           11000
```

Explanation:
In the above example, the data that has to be retrieved is available in the **ACCT_MSTR**, which holds the accounts held by the bank. The output requirements are the **account number**, the **current balance** of that account, the **branch** to which that account belongs and the **average balance** of that branch.The first three requirements can be retrieved from the **ACCT_MSTR** table.

The average of the balance on a per branch basis requires use of another select query and a group by clause. This means a sub query can be used, but in this case, the sub query will return a value, which will be a part of the output. Since this query is going to act as a source of data it is placed in the **FROM** clause of the outer query and given an alias **B**. Finally to produce the output a join is used to get the data on the basis of the outer query i.e. (**A**.BRANCH_NO = **B**.BRANCH_NO) followed by a **WHERE** clause which actually filters the data before producing the output.

To understand the solution the query mentioned above needs to be simplified. The inner most sub-queries should be handled first and then continued outwards.

The **first step** is to identify the branch numbers and their average balance. This is done by, extracting the value held by the **BRANCH_NO** field from the **BRANCH_MSTR** table. The SQL query for this will be as follows:

```
SELECT BRANCH_NO, AVG(CURBAL) AVGBAL FROM ACCT_MSTR
    GROUP BY BRANCH_NO
```

The target table will be as follows:

Output:

```
BRANCH_NO   AVGBAL
---------- --------
B1            7666.67
B2            2833.33
B3                500
B4                500
B5              11000
B6              11000
6 rows selected.
```

The **second step** is to associate the data returned by the inner query with the outer. This is done by binding the Sub-query with the **FROM** clause and using join. The output shown above is treated as an individual (temporary) table. This new table is referred as **B**, the alias name specified in the main **SELECT** statement.

The **third step** is to filter the data to output only those records where the current balance is more then the average balance of the branch to which they belong This is done using a **WHERE** clause i.e.(**A**.CURBAL >**B**.AVGBAL)

Finally, the **SELECT** statement is executed as a JOIN i.e. (**WHERE A**.BRANCH_NO = **B**.BRANCH_NO). This is explained in greater depth later in this chapter.

Using Correlated Sub-queries

A sub-query becomes correlated when the subquery references a column from a table in the parent query. A correlated subquery is evaluated once for each row processed by the parent statement, which can be any of SELECT, DELETE or UPDATE.

A correlated subquery is one way of reading every row in a table and comparing values in each row against related data. It is used whenever a subquery must return a different result for each candidate row considered by the parent query.

Example 14:
List accounts along with the current balance and the branch to which it belongs, having a balance more than the average balance of the branch, to which the account belongs.

Synopsis:

Tables:	ACCT_MSTR
Columns:	ACCT_NO, CURBAL, BRANCH_NO
Technique:	**Sub-Queries, Join, Functions:** AVG(), **Clauses:** WHERE

Solution:

```
SELECT ACCT_NO, CURBAL, BRANCH_NO FROM ACCT_MSTR A
    WHERE CURBAL > (SELECT AVG(CURBAL) FROM ACCT_MSTR
        WHERE BRANCH_NO = A.BRANCH_NO);
```

Output:

```
ACCT_NO   CURBAL     BRANCH_NO
-------------------------------
CA2           3000 B2
CA4          12000 B5
CA7          22000 B1
CA10         32000 B6
CA12          5000 B2
```

Explanation:

In the above example, the data that has to be retrieved is available in the **ACCT_MSTR**, which holds the accounts held by the bank. The output requirements are the **account number**, the **current balance** of that account, the **branch** to which that account belongs. These requirements can be retrieved from the **ACCT_MSTR** table. However the average balance on a per branch basis requires use of another select query.

This means a correlated sub query can be used. The correlated sub-query specifically computes the average balance of each branch. Since both the queries (i.e. Outer and the Inner) use **ACCT_MSTR** table an alias is allotted to the table in the outer query. It is because of this **alias** the inner query is able to distinguish the inner column from the outer column.

Using Multi Column Subquery

Example 15:

Find out all the customers having same names as the employees.

Synopsis:

Tables:	CUST_MSTR, EMP_MSTR
Columns:	CUST_MSTR: FNAME, LNAME, EMP_MSTR: FNAME, LNAME
Technique:	**Sub_Queries, Operators:** IN, **Clauses:** WHERE

Solution:

```
SELECT FNAME, LNAME FROM CUST_MSTR
    WHERE (FNAME, LNAME) IN(SELECT FNAME, LNAME FROM EMP_MSTR);
```

Output:

```
FNAME      LNAME
--------------------------
Ivan       Bayross
```

Explanation:

In the above example, each row of the outer query is compared to the values from the inner query (Multi Row and Multi Column). This means that the values of **FNAME** and **LNAME** from the outer query are compared with **FNAME** and **LNAME** values retrieved by the inner query.

Using Sub-query in CASE Expressions

Example 16:
List the account numbers along with the transaction date, transaction type i.e. whether it's a deposit or withdrawal, the mode of transaction **i.e.** Cash or Cheque and the amount of transaction.

Synopsis:	
Tables:	TRANS_MSTR, TRANS_DTLS
Columns:	TRANS_MSTR: ACCT_NO, DT, DR_CR, TRANS_NO, AMT, TRANS_DTLS: TRANS_NO
Technique:	**Operators:** IN, **Clauses:** CASE WHEN ... THEN

Solution:
```
SELECT ACCT_NO, DT, DR_CR, (CASE WHEN TRANS_NO IN(SELECT TRANS_NO
   FROM TRANS_DTLS) THEN 'Cheque' ELSE 'Cash' END) "Mode", AMT FROM TRANS_MSTR;
```

Output:
```
ACCT_NO    DT        DR_CR   Mode      AMT
-------    --------  -----   -------   ----
SB1        05-NOV-03   D     Cash       500
CA2        10-NOV-03   D     Cash      2000
CA2        13-NOV-03   D     Cash      3000
SB3        22-NOV-03   D     Cash       500
CA2        10-DEC-03   W     Cash      2000
CA4        05-DEC-03   D     Cheque    2000
SB5        15-DEC-03   D     Cheque     500
SB6        27-DEC-03   D     Cash       500
CA7        14-JAN-04   D     Cheque    2000
SB8        29-JAN-04   D     Cash       500
SB9        05-FEB-04   D     Cash       500
SB9        15-FEB-04   D     Cheque    3000
SB9        17-FEB-04   W     Cash      2500
SB9        19-FEB-04   D     Cheque    2000
CA10       05-APR-04   D     Cheque    3000
SB9        27-APR-04   W     Cash      2500
SB9        05-MAR-04   D     Cash       500
SB11       10-MAR-04   D     Cash      2000
CA12       22-MAR-04   D     Cash       500
SB13       05-APR-04   D     Cheque    2000
CA14
20 rows selected.
```

Explanation:
In the above example, the inner query will return a value (i.e. if a record exists in the **TRANS_DTLS** table then the transaction is done via a Cheque). Based on the value returned the outer query will display the Transaction mode as either Cheque or Cash.

Using Subquery In An ORDER BY clause

Example 17:
List the employees of the bank in the order of the branch names at which they are employed.

Synopsis:	
Tables:	EMP_MSTR,
Columns:	EMP_MSTR: EMP_NO, FNAME, LNAME, DEPT
Technique:	**Clauses:** ORDER BY **Others:** Alias, Concat (||)

Solution:

```
SELECT EMP_NO, (FNAME || ' ' || LNAME) "Name", DEPT FROM EMP_MSTR E
    ORDER BY (SELECT NAME FROM BRANCH_MSTR B
        WHERE E.BRANCH_NO = B.BRANCH_NO);
```

Output:

```
EMP_NO    Name              DEPT
------    ----              ----
E2        Amit Desai        Loans And Financing
E9        Vikram Randive    Marketing
E3        Maya Joshi        Client Servicing
E8        Seema Apte        Client Servicing
E6        Sonal Khan        Administration
E10       Anjali Pathak     Administration
E5        Mandhar Dalvi     Marketing
E7        Anil Kambli       Marketing
E1        Ivan Bayross      Administration
E4        Peter Joseph      Loans And Financing
10 rows selected.
```

Explanation:

In the above example, the output needs to be ordered on the basis of branch names in which they are employed. The Data required is available in the **EMP_MSTR** table. Since the output needs to be ordered on the basis of branch names, which are available in the **BRANCH_MSTR** table, there is a need of a separate query, which can return the branch names from the **BRANCH_MSTR** table to which the employees belong. Based on the values returned from the inner query the output produced by the outer query will be ordered. This is done, by placing the inner query, in the **ORDER BY** clause and further correlated with the outer query on the basis of the **BRANCH_NO** being **the common field** in the tables **EMP_MSTR** and **BRANCH_MSTR**.

Using EXISTS / NOT EXISTS Operator

The **EXISTS** operator is usually used with correlated subqueries. This operator enables to test whether a value retrieved by the outer query exists in the results set of the values retrieved by the inner query. If the subquery returns at least one row, the operator returns **TRUE**. If the value does not exist, it returns **FALSE**.

The **EXISTS** operator ensures that the search in the inner query terminates when at least one match is found.

Similarly, the **NOT EXISTS** operator enables to test whether a value retrieved by the outer query is not a part of the result set of the values retrieved by the inner query.

Example 18:

List employees who have verified at least one account.

Synopsis:

Tables:	EMP_MSTR, ACCT_MSTR
Columns:	EMP_MSTR: EMP_NO, FNAME, LNAME, ACCT_MSTR: VERI_EMP_NO
Technique:	**Operators:** EXISTS(), **Clauses:** WHERE

Solution:

```
SELECT EMP_NO, FNAME, LNAME FROM EMP_MSTR E WHERE EXISTS(SELECT 'SCT'
    FROM ACCT_MSTR WHERE VERI_EMP_NO = E.EMP_NO);
```

Output:

```
EMP_NO     FNAME          LNAME
-------------------------------
           Ivan           Bayross
E1         Peter          Joseph
E4
```

Explanation:
In the above example, the inner query is correlated with the outer query via the **EMP_NO** field. As soon as the search in the inner query retrieves at least one match, i.e. **VERI_EMP_NO = E.EMP_NO**, the search is terminated. This means that the inner query stops it's processing and the outer query then produces the output. In the case of the inner query there is no need to return a specific value, hence a constant **'SCT'** is used instead. This is useful in terms of performance as it will be faster to select a constant than a column.

Example 19:
List those branches, which don't have employees yet.

Synopsis:

Tables:	BRANCH_MSTR, EMP_MSTR
Columns:	BRANCH_MSTR: BRANCH_NO, NAME, LNAME, EMP_MSTR: BRANCH_NO
Technique:	Operators: NOT, EXISTS(), Clauses: WHERE Others: Alias

Solution:

```
SELECT BRANCH_NO, NAME FROM BRANCH_MSTR B WHERE NOT EXISTS(SELECT 'SCT'
   FROM EMP_MSTR WHERE BRANCH_NO = B.BRANCH_NO);
```

Output:

```
BRANCH_NO     NAME
-------------------
B5            Borivali
```

Explanation:
In the above example, the inner query is correlated with the outer query via the **BRANCH_NO** field. Since the **NOT EXISTS** operator is used, if the inner query retrieves no rows at all i.e. the condition **BRANCH_NO = B.BRANCH_NO** fails, the outer query produces the output. This means after the inner query stops it's processing, the outer query sends the output based on the operator used. In the case of the inner query there is no need to return a specific value, hence a constant **'SCT'** is used instead. This is useful in terms of performance as it will be faster to select a constant than a column.

JOINS

Joining Multiple Tables (Equi Joins)

Sometimes it is necessary to work with multiple tables as though they were a single entity. Then a single SQL sentence can manipulate data from all the tables. **Joins** are used to achieve this. Tables are joined on columns that have the same **data type** and **data width** in the tables.

Tables in a database can be related to each other with keys. A primary key is a column with a unique value for each row. The purpose is to bind data together, across tables, without repeating all of the data in every table.

The JOIN operator specifies how to relate tables in the query.

Types of JOIN:

- INNER
- OUTER (LEFT, RIGHT, FULL)
- CROSS

INNER JOIN: Inner joins are also known as **Equi Joins**. There are the most common joins used in SQL*Plus. They are know as equi joins because the where statement generally compares two columns from two tables with the equivalence operator =. This type of join is by far the most commonly used. In fact, many systems use this type as the default join. This type of join can be used in situations where selecting only those rows that have values in common in the columns specified in the ON clause, is required. In short, the INNER JOIN returns all rows from both tables where there is a match

OUTER JOIN: Outer joins are similar to inner joins, but give a bit more flexibility when selecting data from related tables. This type of join can be used in situations where it is desired, to select all rows from the table on the left (or right, or both) regardless of whether the other table has values in common and (usually) enter NULL where data is missing.

CROSS JOIN: A cross join returns what's known as a Cartesian product. This means that the join combines every row from the left table with every row in the right table. As can be imagined, sometimes this join produces a mess, but under the right circumstances, it can be very useful. This type of join can be used in situations where it is desired, to select all possible combinations of rows and columns from both tables. This kind of join is usually not preferred as it may run for a very long time and produce a huge result set that may not be useful.

Syntax:

ANSI-style

```
SELECT <ColumnName1>, <ColumnName2>, <ColumnName N> FROM <TableName1>
    INNER JOIN <TableName2>
        ON <TableName1>.<ColumnName1>=<TableName2>.<ColumnName2>
            WHERE <Condition>
        ORDER BY <ColumnName1>, <ColumnName2>, <ColumnNameN>
```

Theta-style

```
SELECT <ColumnName1>, <ColumnName2>, <ColumnNameN>
    FROM <TableName1>,<TableName2>
    WHERE <TableName1>.<ColumnName1> = <TableName2>.<ColumnName2>
        AND <Condition>
        ORDER BY <ColumnName1>, <ColumnName2>, <ColumnNameN>
```

In the above syntax:

- **ColumnName1** in **TableName1** is usually that table's **Primary Key**
- **ColumnName2** in **TableName2** is a **Foreign Key** in that table
- **ColumnName1** and **ColumnName2** must have the **same Data Type** and for certain data types, the same size

Inner Join

Example 20:

List the employee details along with branch names to which they belong.

Synopsis:

Tables:	EMP_MSTR, BRANCH_MSTR
Columns:	EMP_MSTR: EMP_NO, FNAME, MNAME, LNAME, DEPT, DESIG, BRANCH_NO BRANCH_MSTR: NAME, BRANCH_NO
Technique:	**Join:** INNER JOIN ... ON, SIMPLE, **Clauses:** WHERE, **Others:** Concat (||)

Solution 1 (Ansi-style):

```
SELECT E.EMP_NO, (E
    E.DEPT, E.DES
FROM EMP_MSTR
    ON B.BRANCH
```

Solution 2 (Theta-style)

```
SELECT E.EMP_NO,
    E.DEPT, E.DE
FROM EMP_MST
```

Output:

```
EMP Name
NO
--------------------
E1  Ivan Nelson
E4  Peter Iyer
E2  Amit Desai
E9  Vikram Vila
E3  Maya Mahima
E8  Seema P. A
E5  Mandhar Di
E7  Anil Ashut
E6  Sonal Abdu
E10 Anjali Sam
10 rows select
```

Explanation:

In the above exam
no two rows can h
same name. The
BRANCH_MST
in the **EMP_MST**

Notice that:

- The **EMP_N**
- The **BRANC**
- The **BRAN**
  BRANCH_

On the basis of
to link to the **B**
the use of inner

Note

If th
colu

Solution 1 (Ansi-style):

```
SELECT E.EMP_NO, (E.FNAME || ' ' || E.MNAME || ' ' || E.LNAME) "Name", B.NAME "Branch",
       E.DEPT, E.DESIG
  FROM EMP_MSTR E INNER JOIN BRANCH_MSTR B
    ON B.BRANCH_NO = E.BRANCH_NO;
```

Solution 2 (Theta-style):

```
SELECT E.EMP_NO, (E.FNAME || ' ' || E.MNAME || ' ' || E.LNAME) "Name", B.NAME "Branch",
       E.DEPT, E.DESIG
  FROM EMP_MSTR E, BRANCH_MSTR B WHERE B.BRANCH_NO = E.BRANCH_NO;
```

Output:

```
EMP Name                 Branch          DEPT                 DESIG
NO
--- -------------------- --------------- -------------------- --------------------
E1  Ivan Nelson Bayross  Vile Parle (HO) Administration       Managing Director
E4  Peter Iyer Joseph    Vile Parle (HO) Loans And Financing  Clerk
E2  Amit Desai           Andheri         Loans And Financing  Finance Manager
E9  Vikram Vilas Randive Andheri         Marketing            Sales Asst.
E3  Maya Mahima Joshi    Churchgate      Client Servicing     Sales Manager
E8  Seema P. Apte        Churchgate      Client Servicing     Clerk
E5  Mandhar Dilip Dalvi  Mahim           Marketing            Marketing Manager
E7  Anil Ashutosh Kambli Mahim           Marketing            Sales Asst.
E6  Sonal Abdul Khan     Darya Ganj      Administration       Admin. Executive
E10 Anjali Sameer Pathak Darya Ganj      Administration       HR Manager
10 rows selected.
```

Explanation:

In the above example, in the **EMP_MSTR** table, the **EMP_NO** column is the **primary key**, meaning that no two rows can have the same **EMP_NO**. The **EMP_NO** distinguishes two persons even if they have the same name. The data required in this example is available in two tables i.e. **EMP_MSTR** and **BRANCH_MSTR**. This is because branch names are going to be a part of the output but are not available in the **EMP_MSTR** table.

Notice that:

- The **EMP_NO** column is the primary key of the **EMP_MSTR** table
- The **BRANCH_NO** column is the primary key of the **BRANCH_MSTR** table
- The **BRANCH_NO** column in the **EMP_MSTR** table is used to refer to the branches in the **BRANCH_MSTR** table without using their names

On the basis of the reference available in the **EMP_MSTR** table i.e. the **BRANCH_NO** field its possible to link to the **BRANCH_MSTR** table and fetch the Branch names for display. This is easily possible with the use of inner join based on the condition (**B.**BRANCH_NO = **E.**BRANCH_NO).

Note

If the columnnames on which the **join** is to be specified are the same in each table reference the columns using **TableName.ColumnName**.

Example 21:
List the customers along with their multiple address details.

Synopsis:

Tables:	CUST_MSTR, ADDR_DTLS
Columns:	CUST_MSTR: CUST_NO, FNAME, MNAME, LNAME ADDR_DTLS: CODE_NO, ADDR1, ADDR2, CITY, STATE, PINCODE
Technique:	**Join:** INNER JOIN ... ON, SIMPLE, **Clauses:** WHERE **Others:** Concat (||)

Solution 1 (Ansi-style):

```
SELECT C.CUST_NO, (C.FNAME || ' ' || C.MNAME || ' ' || C.LNAME) "Customer", (A.ADDR1 ||''||
        A.ADDR2 ||''|| A.CITY ||','|| A.STATE ||','|| A.PINCODE) "Address"
    FROM CUST_MSTR C INNER JOIN ADDR_DTLS A ON C.CUST_NO = A.CODE_NO
        WHERE C.CUST_NO LIKE 'C%' ORDER BY C.CUST_NO;
```

Solution 2 (Theta-style):

```
SELECT C.CUST_NO, (C.FNAME || ' ' || C.MNAME || ' ' || C.LNAME) "Customer", (A.ADDR1 ||''||
        A.ADDR2 ||''|| A.CITY ||','|| A.STATE ||','|| A.PINCODE) "Address"
    FROM CUST_MSTR C, ADDR_DTLS A WHERE C.CUST_NO = A.CODE_NO
        AND C.CUST_NO LIKE 'C%' ORDER BY C.CUST_NO;
```

Output:

```
CUST_NO Customer
------------------------------------
Address
--------------------------------------------------------------------------------
C1      Ivan Nelson Bayross
F-12, Diamond Palace, West Avenue, North Avenue, Santacruz (West),
Mumbai, Maharashtra, 400056
C10     Namita S. Kanade
B-10, Makarand Society, Cadal Road, Mahim, Mumbai, Maharashtra, 400016
C2      Chriselle Ivan Bayross
F-12, Silver Stream, Santacruz (East), Mumbai, Maharashtra, 400056
C3      Mamta Arvind Muzumdar
Magesh Prasad, Saraswati Baug, Jogeshwari(E), Mumbai, Maharashtra,
400060
C4      Chhaya Sudhakar Bankar
4, Sampada, Kataria Road, Mahim, Mumbai, Maharashtra, 400016
C5      Ashwini Dilip Joshi
104, Vikram Apts. Bhagat Lane, Shivaji Park, Mahim, Mumbai,
Maharashtra, 400016
C6      Hansel I. Colaco
12, Radha Kunj, N.C Kelkar Road, Dadar, Mumbai, Maharashtra, 400028
C6      Hansel I. Colaco
203/A, Prachi Apmt., Andheri (East), Mumbai, Maharashtra, 400058
C7      Anil Arun Dhone
A/14, Shanti Society, Mogal Lane, Mahim, Mumbai, Maharashtra, 400016
C8      Alex Austin Fernandes
5, Vagdevi, Senapati Bapat Rd., Dadar, Mumbai, Maharashtra, 400016
C9      Ashwini Shankar Apte
A-10 Nutan Vaishali, Shivaji Park, Mahim, Mumbai, Maharashtra, 400016
11 rows selected.
```

Explanation:
In the above example, the data required is available in two tables i.e. CUST_MSTR and ADDR_DTLS. Both the tables are linked via a common field. This is because the data is spread across the tables based on a normalization schema.

Notice that:

- The **CUST_NO** column is the primary key of the **CUST_MSTR** table
- The **ADDR_DTLS** is a table that holds the address details of customers.
  In **ADDR_DTLS** table:
  - **ADDR_NO** column is the primary key of that table.
  - **CODE_NO** column is used to refer to the customers in the **CUST_MSTR** table via the **CUST_NO** column

To retrieve the data required, both the tables have to linked on the basis of a common column using joins as follows:

- **C**.CUST_NO = **A**.CODE_NO

This means the CUST_NO field of CUST_MSTR table is joined with CODE_NO field of the ADDR_DTLS table

Now since both the tables are linked using a join, data can be retrieved as if they are all in one table using the alias as:
C.CUST_NO, (**C**.FNAME || ' ' || **C**.MNAME || ' ' || **C**.LNAME) "Customer", (**A**.ADDR1 || ' ' || **A**.ADDR2 || ' ' || **A**.CITY || ', ' || **A**.STATE || ', ' || **A**.PINCODE) "Address"

Finally the output is ordered on the basis of Customers.

Example 22:
List the Customers along with the account details associated with them.

Synopsis:

Tables:	CUST_MSTR, ACCT_MSTR, ACCT_FD_CUST_DTLS, BRANCH_MSTR
Columns:	ACCT_FD_CUST_DTLS: ACCT_FD_NO, CUST_NO CUST_MSTR: CUST_NO, FNAME, MNAME, LNAME ACCT_MSTR: ACCT_NO, BRANCH_NO, CURBAL BRANCH_MSTR: NAME, BRANCH_NO
Technique:	**Join:** INNER JOIN ... ON, **Operators:** ‖ **Join:** SIMPLE, **Operators:** ‖, **Clauses:** WHERE

Solution 1 (Ansi-style):

```
SELECT C.CUST_NO, (C.FNAME || ' ' || C.MNAME || ' ' || C.LNAME) "Customer", A.ACCT_NO,
       B.NAME "Branch", A.CURBAL
  FROM CUST_MSTR C INNER JOIN ACCT_FD_CUST_DTLS L
    ON C.CUST_NO = L.CUST_NO INNER JOIN ACCT_MSTR A
    ON L.ACCT_FD_NO = A.ACCT_NO INNER JOIN BRANCH_MSTR B
    ON A.BRANCH_NO = B.BRANCH_NO ORDER BY C.CUST_NO, A.ACCT_NO;
```

Solution 2 (Theta-style):

```
SELECT C.CUST_NO, (C.FNAME || ' ' || C.MNAME || ' ' || C.LNAME) "Customer", A.ACCT_NO,
       B.NAME "Branch", A.CURBAL
  FROM CUST_MSTR C, ACCT_FD_CUST_DTLS L, ACCT_MSTR A, BRANCH_MSTR B
 WHERE C.CUST_NO = L.CUST_NO AND L.ACCT_FD_NO = A.ACCT_NO
   AND A.BRANCH_NO = B.BRANCH_NO ORDER BY C.CUST_NO, A.ACCT_NO;
```

Output:

CUST_NO	Customer	ACCT_NO	Branch	CURBAL
C1	Ivan Nelson Bayross	SB1	Vile Parle (HO)	500
C1	Ivan Nelson Bayross	SB11	Vile Parle (HO)	500
C1	Ivan Nelson Bayross	SB15	Darya Ganj	500
C1	Ivan Nelson Bayross	SB5	Darya Ganj	32000
C10	Namita S. Kanade	CA10	Mahim	500
C10	Namita S. Kanade	SB9	Andheri	5000
C2	Chriselle Ivan Bayross	CA12	Andheri	3000
C2	Chriselle Ivan Bayross	CA2	Andheri	5000
C3	Mamta Arvind Muzumdar	CA12	Andheri	3000
C3	Mamta Arvind Muzumdar	CA2	Mahim	500
C3	Mamta Arvind Muzumdar	SB9	Borivali	10000
C4	Chhaya Sudhakar Bankar	CA14	Borivali	12000
C4	Chhaya Sudhakar Bankar	CA4	Churchgate	500
C4	Chhaya Sudhakar Bankar	SB13	Darya Ganj	500
C4	Chhaya Sudhakar Bankar	SB15	Churchgate	500
C4	Chhaya Sudhakar Bankar	SB3	Darya Ganj	500
C4	Chhaya Sudhakar Bankar	SB5	Borivali	10000
C4	Chhaya Sudhakar Bankar	CA14	Borivali	12000
C5	Ashwini Dilip Joshi	CA4	Mahim	500
C5	Ashwini Dilip Joshi	SB6	Vile Parle (HO)	22000
C5	Ashwini Dilip Joshi	CA7	Mahim	500
C6	Hansel I. Colaco	SB6	Vile Parle (HO)	22000
C7	Anil Arun Dhone	CA7	Darya Ganj	32000
C8	Alex Austin Fernandes	CA10	Andheri	500
C9	Ashwini Shankar Apte	SB8		
C9	Ashwini Shankar Apte			

25 rows selected.

Explanation:

In the above example, the data required is available in four tables i.e. CUST_MSTR, ACCT_FD_CUST_DTLS, ACCT_MSTR and BRANCH_MSTR. All the four tables are linked via some field in the other table. This is because the data is spread across the tables based on a normalization scheme.

Notice that:

- The **CUST_NO** column is the primary key of the **CUST_MSTR** table
- The **ACCT_NO** column is the primary key of the **ACCT_MSTR** table
- The **ACCT_FD_CUST_DTLS** is a link table between the CUST_MSTR and the ACCT_MSTR table. This table holds information related to which accounts belong to which customers.

  In **ACCT_FD_CUST_DTLS** table:
  - The **ACCT_FD_NO** column is used to refer to the accounts in the **ACCT_MSTR** table via the **ACCT_NO** column
  - The **CUST_NO** column is used to refer to the customers in the **CUST_MSTR** table via the **CUST_NO** column
- The **BRANCH_NO** column is the primary key of the **BRANCH_MSTR** table

To retrieve the data required, all the four tables have to linked on the basis of common columns using joins as follows:

- **C**.CUST_NO = **L**.CUST_NO

This means the CUST_NO field of CUST_MSTR table is joined with CUST_NO field of the ACCT_FD_CUST_DTLS table

- **L.ACCT_FD_NO = A.ACCT_NO**
  This means the ACCT_FD_NO field of ACCT_FD_CUST_DTLS table is joined with ACCT_NO field of the ACCT_MSTR table
- **A.BRANCH_NO = B.BRANCH_NO**
  This means the BRANCH_NO field of ACCT_MSTR table is joined with BRANCH_NO field of the BRANCH_MSTR table

Now since the tables are linked using a join, data can be retrieved as if they are all in one table using the alias as:

C.CUST_NO, (**C.**FNAME || ' ' || **C.**MNAME || ' ' || **C.**LNAME) "Customer", **A.**ACCT_NO, **B.**NAME "Branch", **A.**CURBAL

Finally the output is ordered on the basis of Customers and within Customer, Account numbers.

Adding An Additional WHERE Clause Condition

Example 23:
List the employee details of only those employees who belong to the Administration department along with branch names to which they belong.

Synopsis:

Tables:	EMP_MSTR, BRANCH_MSTR
Columns:	EMP_MSTR: EMP_NO, FNAME, MNAME, LNAME, DEPT, DESIG, BRANCH_NO BRANCH_MSTR: NAME, BRANCH_NO
Technique:	**Join:** INNER JOIN ... ON, **Clauses:** WHERE, **Others:** Alias

Solution:

```
SELECT E.EMP_NO, (E.FNAME || ' ' || E.MNAME || ' ' || E.LNAME) "Name", B.NAME "Branch",
       E.DEPT, E.DESIG
  FROM EMP_MSTR E INNER JOIN BRANCH_MSTR B ON B.BRANCH_NO = E.BRANCH_NO
    WHERE E.DEPT = 'Administration';
```

Output:

```
EMP_NO  Name                  Branch          DEPT            DESIG
------  --------------------  --------------  --------------  ----------------
E1      Ivan Nelson Bayross   Vile Parle (HO) Administration  Managing Director
E6      Sonal Abdul Khan      Darya Ganj      Administration  Admin. Executive
E10     Anjali Sameer Pathak  Darya Ganj      Administration  HR Manager
```

Explanation:
In the above example, the data required is available in two tables i.e. EMP_MSTR and BRANCH_MSTR. Both the tables are linked via a common field. This is because the data is spread across the tables based on a normalization schema.

Notice that:

- The **EMP_NO** column is the primary key of the **EMP_MSTR** table
- The **BRANCH_NO** column is used to refer to the branch names in the BRANCH_MSTR table via the BRANCH_NO column
- The **BRANCH_NO** column is the primary key of the **BRANCH_MSTR** table

To retrieve the data required, both the tables have to linked on the basis of a common column using joins as follows:

- **B.**BRANCH_NO = **E.**BRANCH_NO

This means the BRANCH_NO field of BRANCH_MSTR table is joined with BRANCH_NO field of the EMP_MSTR table

Now since both the tables are linked using a join, data can be retrieved as if they are all in one table using the alias as:
E.EMP_NO, (**E.**FNAME || ' ' || **E.**MNAME || ' ' || **E.**LNAME) "Name", **B.**NAME "Branch", **E.**DEPT, **E.**DESIG

Finally the output is filtered to display only those employees belonging to the administration department using WHERE clause as:
E.DEPT = 'Administration'

Outer Join

Example 24:
List the employee details along with the contact details (if any) Using Left Outer Join.

Synopsis:

Tables:	EMP_MSTR, CNTC_DTLS
Columns:	EMP_MSTR: EMP_NO, FNAME, LNAME, DEPT CNTC_DTLS: CODE_NO, CNTC_TYPE, CNTC_DATA
Technique:	**Join:** LEFT JOIN ... ON, **Clauses:** WHERE **Others:** Alias, Concat (||)

Solution 1 (Ansi-style):
```
SELECT (E.FNAME || ' ' || E.LNAME) "Name", E.DEPT, C.CNTC_TYPE, C.CNTC_DATA
    FROM EMP_MSTR E LEFT JOIN CNTC_DTLS C ON E.EMP_NO = C.CODE_NO;
```

Solution 2 (Theta-style):
```
SELECT (E.FNAME || ' ' || E.LNAME) "Name", E.DEPT, C.CNTC_TYPE, C.CNTC_DATA
    FROM EMP_MSTR E, CNTC_DTLS C WHERE E.EMP_NO = C.CODE_NO(+);
```

Output:
```
Name            DEPT                    CNTC_TYPE    CNTC_DATA
Amit Desai      Loans And Financing     R            28883779
Maya Joshi      Client Servicing        R            28377634
Peter Joseph    Loans And Financing     R            26323560
Mandhar Dalvi   Marketing               R            26793231
Sonal Khan      -Administration         R            28085654
Anil Kambli     Marketing               R            24442342
Seema Apte      Client Servicing        R            24365672
Vikram Randive  Marketing               R            24327349
Anjali Pathak   Administration          R            24302579
Ivan Bayross    Administration
10 rows selected.
```

Explanation:
In the above example, the data required is all the employee details along with their contact details if any. This means all the employee details have to be listed even though their corresponding contact information is not present. The data is available in two tables i.e. EMP_MSTR and CNTC_DTLS

In such a situation, the **LEFT JOIN** can be used which returns all the rows from the first table (i.e. EMP_MSTR), even if there are no matches in the second table (CNTC_DTLS). This means, if there are employees in **EMP_MSTR** that do not have any contacts in **CNTC_DTLS**, those rows will also be listed. Notice the keyword **LEFT JOIN** in the first solution (Ansi-style) and the (+) in the second solution (Theta-style). This indicates that all rows from the first table i.e. EMP_MSTR will be displayed even though there exists no matching rows in the second table i.e. CNTC_DTLS.

Notice that:

- The **EMP_NO** column is the primary key of the **EMP_MSTR** table
- The **CNTC_DTLS** is a table that holds the contact details of employees.
- In **CNTC_DTLS** table:
  - **ADDR_NO** column is used to refer to the addresses in the ADDR_DTLS table via the ADDR_NO column.
  - **CODE_NO** column is used to refer to the employees in the **EMP_MSTR** table via the **EMP_NO** column

To retrieve the data required, both the tables have to linked on the basis of common columns using joins as follows:

- **E**.EMP_NO = **C**.CODE_NO

This means the EMP_NO field of EMP_MSTR table is joined with CODE_NO field of the CNTC_DTLS table

Example 25:
List the employee details along with the contact details (if any) Using Right Outer Join.

Synopsis:

Tables:	EMP_MSTR, CNTC_DTLS
Columns:	EMP_MSTR: EMP_NO, FNAME, LNAME, DEPT CNTC_DTLS: CODE_NO, CNTC_TYPE, CNTC_DATA
Technique:	**Join:** RIGHT JOIN ... ON, **Clauses:** WHERE

Solution 1 (Ansi-style):

```
SELECT E.FNAME, E.LNAME, E.DEPT, C.CNTC_TYPE, C.CNTC_DATA FROM CNTC_DTLS C
   RIGHT JOIN EMP_MSTR E ON C.CODE_NO = E.EMP_NO;
```

Solution 2 (Theta-style):

```
SELECT E.FNAME, E.LNAME, E.DEPT, C.CNTC_TYPE, C.CNTC_DATA
   FROM CNTC_DTLS C, EMP_MSTR E WHERE C.CODE_NO(+) = E.EMP_NO;
```

Output:

```
FNAME      LNAME      DEPT                 C CNTC_DATA
---------- ---------- -------------------- - ----------
Ivan       Bayross    Administration       R 26045953
Anjali     Pathak     Administration       R 24302579
Amit       Desai      Loans And Financing  R 28883779
Maya       Joshi      Client Servicing     R 28377634
Peter      Joseph     Loans And Financing  R 26323560
Mandhar    Dalvi      Marketing            R 26793231
Sonal      Khan       Administration       R 28085654
Anil       Kambli     Marketing            R 24442342
Seema      Apte       Client Servicing     R 24365672
Vikram     Randive    Marketing            R 24327349
10 rows selected.
```

Explanation:
In the above example, the data required is all the employee details along with their contact details if any. But in this case **RIGHT JOIN** is being used. This means all the employee details have to be listed even though their corresponding contact information is not present. The data is available in two tables i.e. EMP_MSTR and CNTC_DTLS

Since the **RIGHT JOIN** returns all the rows from the second table even if there are no matches in the first table. The first table in the FROM clause will have to be **CNTC_DTLS** and the second table **EMP_MSTR**. This means, if there are employees in **EMP_MSTR** that do not have any contacts in **CNTC_DTLS**, those rows will also be listed. Notice the keyword **RIGHT JOIN** in the first solution (Ansi-style) and the **(+)** in the second solution (Theta-style). This indicates that all rows from the second table i.e. EMP_MSTR will be displayed even though there exists no matching rows in the first table i.e. CNTC_DTLS.

Notice that:

- The **EMP_NO** column is the primary key of the **EMP_MSTR** table
- The **CNTC_DTLS** is a table that holds the contact details of employees.
  In **CNTC_DTLS** table:
  - **ADDR_NO** column is used to refer to the addresses in the ADDR_DTLS table via the ADDR_NO column.
  - **CODE_NO** column is used to refer to the employees in the **EMP_MSTR** table via the **EMP_NO** column

To retrieve the data required, both the tables have to linked on the basis of common columns using joins as follows:

- **C**.CODE_NO = **E**.EMP_NO

This means the CODE_NO field of CNTC_DTLS table is joined with EMP_NO field of the EMP_MSTR table

Cross Join

Suppose it is desired to combine each deposit amount with a Fixed Deposits Interest Slab table so as to analyze each deposit amount at each interest rate and their minimum and maximum periods. This is elegantly done using a cross join.

Example 26:
Create a report using **cross join** that will display the maturity amounts for pre-defined deposits, based on the Minimum and Maximum periods fixed / time deposits. Ensure that a temporary table called TMP_FD_AMT stores the amounts for pre-defined deposits in the FD_AMT column.

Synopsis:

Tables:	TMP_FD_AMT, FDSLAB_MSTR
Columns:	TMP_FD_AMT: FD_AMT, FDSLAB_MSTR: MINPERIOD, MAXPERIOD, INTRATE
Technique:	**Join:** CROSS JOIN, **Operators:** (*), (/)

Prior executing the SQL statement a table called TMP_FD_AMT has to be created and filled in with some sample data.

```
CREATE TABLE "DBA_BANKSYS"."TMP_FD_AMT"("FD_AMT" NUMBER(6));
```

Insert Statements for the table TMP_FD_AMT:

```
INSERT INTO TMP_FD_AMT (FD_AMT) VALUES(5000);
INSERT INTO TMP_FD_AMT (FD_AMT) VALUES(10000);
INSERT INTO TMP_FD_AMT (FD_AMT) VALUES(15000);
INSERT INTO TMP_FD_AMT (FD_AMT) VALUES(20000);
INSERT INTO TMP_FD_AMT (FD_AMT) VALUES(25000);
INSERT INTO TMP_FD_AMT (FD_AMT) VALUES(30000);
INSERT INTO TMP_FD_AMT (FD_AMT) VALUES(40000);
INSERT INTO TMP_FD_AMT (FD_AMT) VALUES(50000);
```

Solution:

```
SELECT T.FD_AMT, S.MINPERIOD, S.MAXPERIOD, S.INTRATE,
   ROUND(T.FD_AMT + (T.FD_AMT * (S.INTRATE/100) * (S.MINPERIOD/365)))
      "Amt. Min. Period",
ROUND(T.FD_AMT + (T.FD_AMT * (S.INTRATE/100) * (S.MAXPERIOD/365)))
      "Amt. Max. Period"
FROM FDSLAB_MSTR S CROSS JOIN TMP_FD_AMT T;
```

Output:

FD_AMT	MINPERIOD	MAXPERIOD	INTRATE	Amt. Min. Period	Amt. Max. Period
5000	1	30	5	5001	5021
5000	31	92	5.5	5023	5069
5000	93	183	6	5076	5050
. .					
20000	31	92	5.5	20093	20277
20000	93	183	6	20306	20602
20000	184	365	6.5	20655	21300
20000	366	731	7.5	21504	23004
. .					
50000	184	365	6.5	51638	53250
50000	366	731	7.5	53760	57510
50000	732	1097	8.5	58523	62773
50000	1098	1829	10	65041	75055

56 rows selected.

Explanation:

In the above example, the data required is available in two tables i.e. TMP_FD_AMT and FDSLAB_MSTR. In the table TMP_FD_AMT, there exists, the deposit amounts. In the second table FDSLAB_MSTR, there exists a list of fixed deposits slabs comprising of minimum and maximum periods and the interest rates applicable for those periods.

The output is required in the form of a report, which will display calculation based on the FDSLAB_MSTR table for each row held in the TMP_FD_AMT. In such a situation, a CROSS JOIN can be-used which will combine each record from the left table with that of the right table. In this example a cross join will combine each deposit amount (i.e. FD_AMT) from the TMP_FD_AMT table with each slab i.e. each record in the FDSLAB_MSTR table after applying some calculations. Using Cross Join, a matrix between the temporary table named TMP_FD_AMT table and the FDSLAB_MSTR table can be created.

The above SELECT statement creates a record for each deposit amount with the calculated maturity amount based on the interest rates and the minimum and maximum periods. The results are known as a Cartesian product, which combines every record in the left table i.e. FDSLAB_MSTR with every record in the right table i.e. TMP_FD_AMT.

Oracle versions **prior to 9i** don't support an explicit cross join, but the same results can be obtained by using the following statement:

```
SELECT T.FD_AMT, S.MINPERIOD, S.MAXPERIOD, S.INTRATE,
    ROUND(T.FD_AMT + (T.FD_AMT * (S.INTRATE/100) * (S.MINPERIOD/365)))
        "Amt. Min. Period",
    ROUND(T.FD_AMT + (T.FD_AMT * (S.INTRATE/100) * (S.MAXPERIOD/365)))
        "Amt. Max. Period"
    FROM FDSLAB_MSTR S, TMP_FD_AMT T;
```

Guidelines for Creating Joins

- When writing a select statement that joins tables, precede the column name with the table name for clarity
- If the same column name appears in more than one table, the column name must be prefixed with the table name
- The WHERE clause, is the most critical clause in a join select statement. Always make sure to include the WHERE clause

Joining A Table To Itself (Self Joins)

In some situations, it is necessary to join a table to itself, as though joining two separate tables. This is referred to as a **self-join**. In a self-join, two rows from the same table combine to form a result row.

To join a table to itself, **two** copies of the very same table have to be opened in memory. Hence in the **FROM** clause, the table name needs to be mentioned twice. Since the table names are the same, the second table will overwrite the first table and in effect, result in only one table being in memory. This is because a table name is translated into a specific memory location. To avoid this, each table is opened using an alias. Now these table aliases will cause two identical tables to be opened in different memory locations. This will result in two identical tables to be physically present in the computer's memory.

Using the table alias names these two identical tables can be joined.

```
FROM <TableName> [<Alias1>], <TableName> [<Alias2>] ....
```

Example 27:
Retrieve the names of the employees and the names of their respective managers from the employee table.

Synopsis:

Tables:	EMP_MNGR
Columns:	FNAME
Technique:	**Joins:** SELF, **Clauses**: WHERE **Others:** Alias

Solution:

```
SELECT EMP.FNAME "Employee", MNGR.FNAME "Manager"
    FROM EMP_MSTR EMP, EMP_MSTR MNGR
        WHERE EMP.MNGR_NO = MNGR.EMP_NO;
```

Note

In this query, the **EMP_MSTR** table is treated as two separate tables named **EMP** and **MNGR**, using the table **alias** feature of SQL.

Output:

```
Employee
Peter
Sonal
Anil
Seema
Vikram
Anjali
6 rows selec
```

Explanation:
In the above exam
whom they repo
employee numbe

The table EMP_
Emp No
E1
E2
E3
E4
E5
E6
E7
E8
E9
E10

As can be seen
manager (emplo
Similarly, empl
E3, E7, E1 resp

This means:
- The EMP_
- The MNG
  via the EM

This simply m
the table.
From the data
to which the e
manager is als
using a SELF
get the employ

To form a cop
FROM E
Here the EMI
EMP_MSTR

Output:

```
Employee    Manager
----------  ----------
Peter       Amit
Sonal       Ivan
Anil        Mandhar
Seema       Maya
Vikram      Anil
Anjali      Ivan
6 rows selected.
```

Explanation:

In the above example, the data required are all the employees and the names of their respective managers to whom they report. This data is available in the table **EMP_MSTR**. The EMP_MSTR table holds the employee number, their names and the manager numbers who in turn are employees in the same table.

The table EMP_MSTR holds the following data:

Emp_No	Fname	Lname	Mngr_No
E1	Ivan	Bayross	
E2	Amit	Desai	
E3	Maya	Joshi	
E4	**Peter**	**Joseph**	**E2**
E5	Mandhar	Dalvi	
E6	**Sonal**	**Khan**	**E1**
E7	**Anil**	**Kambli**	**E5**
E8	**Seema**	**Apte**	**E3**
E9	**Vikram**	**Randive**	**E7**
E10	**Anjali**	**Pathak**	**E1**

As can be seen from the data above employees named **Peter** having employee number **E4** reports to a manager (employee) named **Amit** having employee number **E2**.
Similarly, employees numbered **E6, E7, E8, E9, E10** report to managers (employees) numbered **E1, E5, E3, E7, E1** respectively.

This means:

- The **EMP_NO** column is the primary key of the **EMP_MSTR** table
- The **MNGR_NO** column is used to refer to the Employee details in the same table i.e. **EMP_MSTR** via the **EMP_NO** column

This simply means that MNGR_NO is a foreign key mapping to the EMP_NO which is the primary key of the table.
From the data available in the EMP_MSTR table seen above, it is possible to extract the manager number to which the employee reports, but in order to extract the manager name i.e. the employee's name (since the manager is also an employee) a reference to the same table EMP_MSTR has to be made. This can be done using a SELF JOIN i.e. making a copy of the same table EMP_MSTR and then referring to the columns to get the employee name against the manager number.

To form a copy of the same table alias have to be used in the FROM clause as:

FROM EMP_MSTR **EMP**, EMP_MSTR **MNGR**

Here the **EMP** is the **first copy** of the table **EMP_MSTR** and **MNGR** is the **second copy** of the table **EMP_MSTR**.

To retrieve the data required, both the copies of the same tables have to linked on the basis of common columns using joins as follows:

- **EMP**.MNGR_NO = **MNGR**.EMP_NO
  This means the MNGR_NO field of EMP_MSTR table (First Copy: EMP) is joined with EMP_NO field of the EMP_MSTR (Second Copy: MNGR) table

CONCATENATING DATA FROM TABLE COLUMNS

Example 28:
Create an English sentence, by joining predetermined string values with column data retrieved from the **ACCT_MSTR** table.

The string literals are:

Account No.	was introduced by Customer No.	At Branch No.

The columns are:

ACCT_NO	INTRO_CUST_NO	BRANCH_NO

Synopsis:

Tables:	ACCT_MSTR
Columns:	ACCT_NO, INTRO_CUST_NO, BRANCH_NO
Technique:	**Other:** Concat (||)

Solution:

```
SELECT 'ACCOUNT NO.' || ACCT_NO || ' WAS INTRODUCED BY CUSTOMER NO.'
    || INTRO_CUST_NO || ' AT BRANCH NO.' || BRANCH_NO FROM ACCT_MSTR;
```

Problem:
Since the above SELECT cannot find an appropriate column header to print on the VDU screen, the SELECT uses the **formula (i.e.** the entire SELECT content) as the column header as described below.

Output:

```
'ACCOUNTNO.'||ACCT_NO||'WASINTRODUCEDBYCUSTOMERNO.'||INTRO_CUST_NO
||'ATBRANCHNO.
--------------------------------------------------------------------------------
ACCOUNT NO. SB1 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B1
ACCOUNT NO. CA2 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B2
ACCOUNT NO. SB3 WAS INTRODUCED BY CUSTOMER NO. C4 AT BRANCH NO. B3
ACCOUNT NO. CA4 WAS INTRODUCED BY CUSTOMER NO. C4 AT BRANCH NO. B5
ACCOUNT NO. SB5 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B6
ACCOUNT NO. SB6 WAS INTRODUCED BY CUSTOMER NO. C5 AT BRANCH NO. B4
ACCOUNT NO. CA7 WAS INTRODUCED BY CUSTOMER NO. C8 AT BRANCH NO. B1
ACCOUNT NO. SB8 WAS INTRODUCED BY CUSTOMER NO. C9 AT BRANCH NO. B2
ACCOUNT NO. SB9 WAS INTRODUCED BY CUSTOMER NO. C10 AT BRANCH NO. B4
ACCOUNT NO. CA10 WAS INTRODUCED BY CUSTOMER NO. C10 AT BRANCH NO. B6
ACCOUNT NO. SB11 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B1
ACCOUNT NO. CA12 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B2
ACCOUNT NO. SB13 WAS INTRODUCED BY CUSTOMER NO. C4 AT BRANCH NO. B3
ACCOUNT NO. CA14 WAS INTRODUCED BY CUSTOMER NO. C4 AT BRANCH NO. B5
ACCOUNT NO. SB15 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B6
15 rows selected.
```

To avoid a data header that appears meaningless, use an **alias** as shown below:

```
SELECT 'ACCOUNT NO. ' || ACCT_NO || ' WAS INTRODUCED BY CUSTOMER NO. '
    || INTRO_CUST_NO || ' AT BRANCH NO. ' || BRANCH_NO "Accounts Opened"
    FROM ACCT_MSTR;
```

Output:

```
Accounts Opened
--------------------------------------------------------------------------------
ACCOUNT NO. SB1 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B1
ACCOUNT NO. CA2 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B2
ACCOUNT NO. SB3 WAS INTRODUCED BY CUSTOMER NO. C4 AT BRANCH NO. B3
ACCOUNT NO. CA4 WAS INTRODUCED BY CUSTOMER NO. C4 AT BRANCH NO. B5
ACCOUNT NO. SB5 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B6
ACCOUNT NO. SB6 WAS INTRODUCED BY CUSTOMER NO. C5 AT BRANCH NO. B4
ACCOUNT NO. CA7 WAS INTRODUCED BY CUSTOMER NO. C8 AT BRANCH NO. B1
ACCOUNT NO. SB8 WAS INTRODUCED BY CUSTOMER NO. C9 AT BRANCH NO. B2
ACCOUNT NO. SB9 WAS INTRODUCED BY CUSTOMER NO. C10 AT BRANCH NO. B4
ACCOUNT NO. CA10 WAS INTRODUCED BY CUSTOMER NO. C10 AT BRANCH NO. B6
ACCOUNT NO. SB11 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B1
ACCOUNT NO. CA12 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B2
ACCOUNT NO. SB13 WAS INTRODUCED BY CUSTOMER NO. C4 AT BRANCH NO. B3
ACCOUNT NO. CA14 WAS INTRODUCED BY CUSTOMER NO. C4 AT BRANCH NO. B5
ACCOUNT NO. SB15 WAS INTRODUCED BY CUSTOMER NO. C1 AT BRANCH NO. B6
15 rows selected.
```

USING THE UNION, INTERSECT AND MINUS CLAUSE

Union Clause

Multiple queries can be put together and their output can be combined using the **union** clause. The **Union** clause merges the output of two or more queries into a single set of rows and columns.

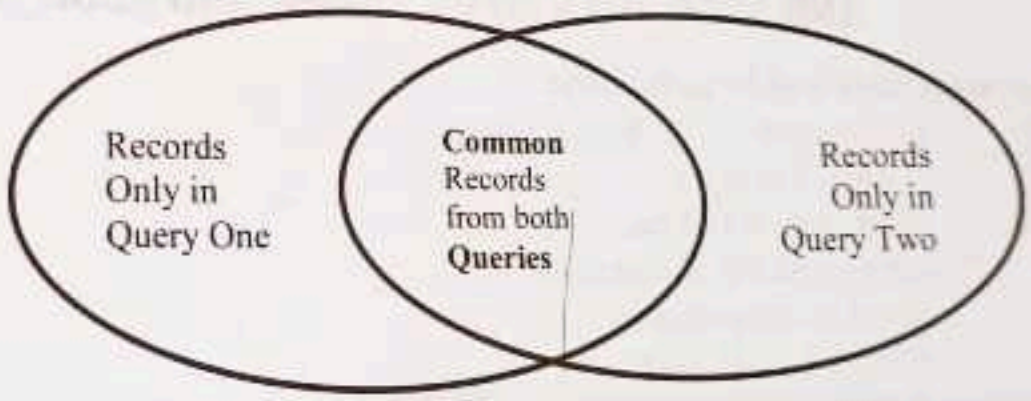

Diagram 10.1: Output of the Union Clause.

Note

The output of both the queries will be as displayed above. The final output of the union clause will be: **Output** = Records from query one + records from query two +
A single set of records, common in both queries.

While working with the UNION clause the following pointers should be considered:

- The number of columns and the data types of the columns being selected must be identical in all the **SELECT** statement used in the query. The names of the columns need not be identical.
- **UNION** operates over all of the columns being selected.
- **NULL** values are not ignored during duplicate checking.
- The **IN** operator has a higher precedence than the **UNION** operator.
- By default, the output is sorted in ascending order of the first column of the **SELECT** clause.

Example 29:
Retrieve the names of all the customers and employees residing in the city of **Mumbai**.

Synopsis:

Tables:	CUST_NO, FNAME, LNAME
Columns:	CUST_MSTR, EMP_MSTR, ADDR_DTLS
Technique:	**Operators:** LIKE, **Clauses:** WHERE, UNION, **Others:** Alias

Solution:
```
SELECT CUST_NO "ID", FNAME || ' ' || LNAME "Customers / Employees"
    FROM CUST_MSTR, ADDR_DTLS
        WHERE CUST_MSTR.CUST_NO = ADDR_DTLS.CODE_NO
            AND ADDR_DTLS.CITY = 'Mumbai' AND ADDR_DTLS.CODE_NO LIKE 'C%'
UNION
SELECT EMP_NO "ID", FNAME || ' ' || LNAME "Customers / Employees"
    FROM EMP_MSTR, ADDR_DTLS
        WHERE EMP_MSTR.EMP_NO = ADDR_DTLS.CODE_NO
            AND ADDR_DTLS.CITY = 'Mumbai' AND ADDR_DTLS.CODE_NO LIKE 'E%';
```

Explanation:
Oracle executes the queries as follows:

The first query in the **UNION** example is as follows:
```
SELECT CUST_NO "ID", FNAME || ' ' || LNAME "Customers / Employees"
    FROM CUST_MSTR, ADDR_DTLS
        WHERE CUST_MSTR.CUST_NO = ADDR_DTLS.CODE_NO
            AND ADDR_DTLS.CITY = 'Mumbai' AND ADDR_DTLS.CODE_NO LIKE 'C%';
```

The target table will be as follows:
```
ID      Customers / Employees
------  ---------------------
C1      Ivan Bayross
C10     Namita Kanade
C2      Chriselle Bayross
C3      Mamta Muzumdar
C4      Chhaya Bankar
C5      Ashwini Joshi
C6      Hansel Colaco
C7      Anil Dhone
C8      Alex Fernandes
C9      Ashwini Apte
10 rows selected.
```

```
SELECT EMP_NO "ID", FNAME || ' ' || LNAME "Customers / Employees"
    FROM EMP_MSTR, ADDR_DTLS WHERE EMP_MSTR.EMP_NO = ADDR_DTLS.CODE_NO
        AND ADDR_DTLS.CITY = 'Mumbai' AND ADDR_DTLS.CODE_NO LIKE 'E%';
```

The target table will be as follows:
```
ID      Customers / Employees
------  ---------------------
E1      Ivan Bayross
E2      Amit Desai
E3      Maya Joshi
E4      Peter Joseph
```

The target table: (Continued)

```
ID      Customers / Employees
E5      Mandhar Dalvi
E6      Sonal Khan
E7      Anil Kambli
E8      Seema Apte
E9      Vikram Randive
9 rows selected.
```

The **UNION** clause picks up the common records as well as the individual records in both queries. Thus, the output after applying the **UNION** clause will be:

Output:

```
ID      Customers / Employees
C1      Ivan Bayross
C10     Namita Kanade
C2      Chriselle Bayross
C3      Mamta Muzumdar
C4      Chhaya Bankar
C5      Ashwini Joshi
C6      Hansel Colaco
C7      Anil Dhone
C8      Alex Fernandes
C9      Ashwini Apte
E1      Ivan Bayross
E2      Amit Desai
E3      Maya Joshi
E4      Peter Joseph
E5      Mandhar Dalvi
E6      Sonal Khan
E7      Anil Kambli
E8      Seema Apte
E9      Vikram Randive
19 rows selected.
```

The Restrictions on using a union are as follows:

- Number of columns in all the queries should be the same
- The data type of the columns in each query must be same
- Unions cannot be used in subqueries
- Aggregate functions cannot be used with union clause

Note

The alias assigned to the first query will be applied in the final output even though an alias has been assigned to the second query it is not applicable.

Intersect Clause

Multiple queries can be put together and their output combined using the intersect clause. The **Intersect** clause outputs only rows produced by **both** the queries intersected **i.e.** the output in an Intersect clause will include only those rows that are retrieved common to both the queries.

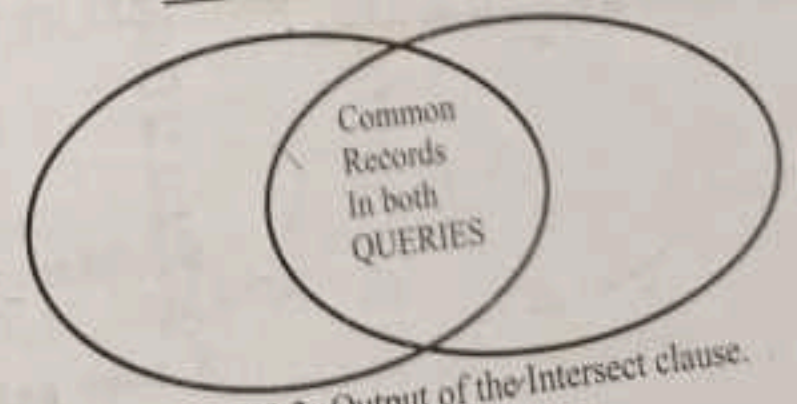

Diagram 10.2: Output of the Intersect clause.

Note

The output of both the queries will be as displayed above. The final output of the Intersect clause will be: **Output** = A single set of records which are common in both queries.

While working with the INTERSECT clause the following pointers should be considered:

- The number of columns and the data types of the columns being selected by the **SELECT** statement in the queries must be identical in all the **SELECT** statements used in the query. The names of the columns need not be identical.
- Reversing the order of the intersected tables does not alter the result.
- **INTERSECT** does not ignore **NULL** values.

Example 30:
Retrieve the customers holding accounts as well as fixed deposits in a bank.

Synopsis:

Tables:	ACCT_FD_CUST_DTLS
Columns:	CUST_NO
Technique:	**Operators:** LIKE, **Clauses:** WHERE, INTERSECT

Solution:

```
SELECT DISTINCT CUST_NO FROM ACCT_FD_CUST_DTLS WHERE ACCT_FD_NO
    LIKE 'CA%' OR ACCT_FD_NO LIKE 'SB%'
INTERSECT
SELECT DISTINCT CUST_NO FROM ACCT_FD_CUST_DTLS
    WHERE ACCT_FD_NO LIKE 'FS%';
```

Explanation:
Oracle executes the queries as follows:

The first query in the **INTERSECT** example is as follows:

```
SELECT DISTINCT CUST_NO FROM ACCT_FD_CUST_DTLS
    WHERE ACCT_FD_NO LIKE 'CA%'
        OR ACCT_FD_NO LIKE 'SB%';
```

The target table will be as follows:

```
CUST_NO
-----------
C1
C10
C2
C3
C4
C5
C6
C7
C8
C9
10 rows selected.
```

The second query in the INTERSECT example is as follows:

```
SELECT DISTINCT CUST_NO FROM ACCT_FD_CUST_DTLS
    WHERE ACCT_FD_NO LIKE 'FS%';
```

The target table will be as follows:

```
CUST_NO
-----------
C10
C2
C3
C4
C5
C6
C8
C9
8 rows selected.
```

The **INTERSECT** clause picks up records that are common in both queries. Thus, the output after applying the INTERSECT clause will be as shown in the output.

Output:

```
CUST_NO
-----------
C10
C2
C3
C4
C5
C6
C8
C9
8 rows selected.
```

Minus Clause

Multiple queries can be pu
output combined using the
Minus clause outputs the r
first query, after **filtering** t
the second query.

Note

The output of bo
be: **Output** = Re

While working with the M
- The number of colum
the queries must be
columns need not be
- All on the columns i
work.

Example 31:
Retrieve the customers h

Synopsis:

Tables:	ACCT
Columns:	CUS
Technique:	Oper

Solution:

```
SELECT DISTINCT
    WHERE ACCT_
MINUS
SELECT DISTINCT
    WHERE ACCT
```

Explanation:
Oracle executes the c

The first query in the
```
SELECT DISTINC
    WHERE ACC
        OR ACCT
```

The second query i
```
SELECT DISTIN
    WHERE ACC
```

Minus Clause

Multiple queries can be put together and their output combined using the minus clause. The **Minus** clause outputs the rows produced by the first query, after **filtering** the rows retrieved by the second query.

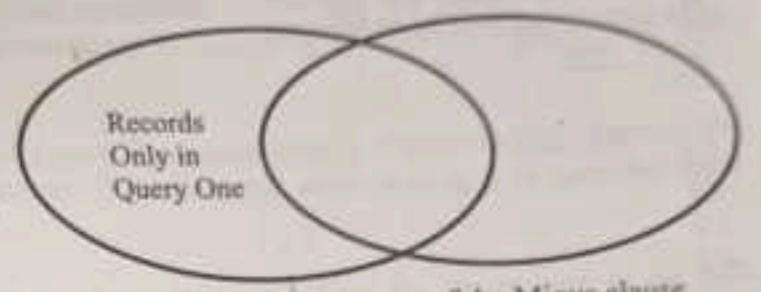

Diagram 10.3: Output of the Minus clause.

Note

The output of both the queries will be as displayed above. The final output of the minus clause will be: **Output** = Records only in query one

While working with the MINUS clause the following pointers should be considered:

- The number of columns and the data types of the columns being selected by the **SELECT** statement in the queries must be identical in all the **SELECT** statements used in the query. The names of the columns need not be identical.
- All on the columns in the **WHERE** clause must be in the **SELECT** clause for the **MINUS** operator to work.

Example 31:
Retrieve the customers holding accounts but not holding any fixed deposits in a bank.

Synopsis:

Tables:	ACCT_FD_CUST_DTLS
Columns:	CUST_NO
Technique:	**Operators:** LIKE, **Clauses:** WHERE, MINUS

Solution:

```
SELECT DISTINCT CUST_NO FROM ACCT_FD_CUST_DTLS
    WHERE ACCT_FD_NO LIKE 'CA%' OR ACCT_FD_NO LIKE 'SB%'
MINUS
SELECT DISTINCT CUST_NO FROM ACCT_FD_CUST_DTLS
    WHERE ACCT_FD_NO LIKE 'FS%';
```

Explanation:
Oracle executes the queries as follows:

The first query in the **INTERSECT** example is as follows:

```
SELECT DISTINCT CUST_NO FROM ACCT_FD_CUST_DTLS
    WHERE ACCT_FD_NO LIKE 'CA%'
        OR ACCT_FD_NO LIKE 'SB%';
```

The second query in the **INTERSECT** example is as follows:

```
SELECT DISTINCT CUST_NO FROM ACCT_FD_CUST_DTLS
    WHERE ACCT_FD_NO LIKE 'FS%';
```

The target table will be as follows:

```
CUST_NO
---------
C1
C10
C2
C3
C4
C5
C6
C7
C8
C9
10 rows selected.
```

The target table will be as follows:

```
CUST_NO
-------
C10
C2
C3
C4
C5
C6
C8
C9

8 rows selected.
```

The **MINUS** clause picks up records in the first query after filtering the records retrieved by the second query. Thus, the output after applying the MINUS clause will be as shown below.

Output:

```
CUST_NO
-------
C1
C7
```

DYNAMIC SQL

When a commercial application is created, its code spec knows exactly what to do, simply because such programmer created code spec is written to do a specific job. For example, a PL/SQL program can prompt a user for:

- An account number
- An amount to be withdrawn
- Then update record data in both the Account and Transaction tables

The PL/SQL code block knows the,

- Tables on which SQL queries will be fired
- Data type of each column
- Constraints defined for each table and column
- Which columns must be updated

prior the program being complied and executed.

However, there are PL/SQL programs that require accepting and processing a variety of SQL statements at their run time (i.e. specifically when they execute).

For example, a program that generates reports may have to manipulate data held within different columns belonging to multiple tables for a specific report.

If the PL/SQL report generation program is to be generic, its SQL statements would not be known before the program is compiled and executed. These SQL statements will only be known at the time when the PL/SQL block runs. This means each time the program is executed, a different SQL statement may be fired against the Db engine depending on specific runtime criteria.

SQL statements that change from a specific runtime execution to another are called dynamic SQL statements. Dynamic SQL statement must adjust themselves to changing application requirements each time.

Dynamic SQL is an enhanced form of Structured Query Language [SQL]. Dynamic SQL permits the constructing of SQL statements dynamically at runtime. This is useful when writing code spec that must adjust to varying databases, conditions or servers.

Dynamic SQL allows creating general purpose, flexible programs where the query text of an SQL statement is unknown at the time of creation, but will be created dynamically and used at compilation and execution time.

WHEN TO CONSIDER USING DYNAMIC SQL

Dynamic SQL should be considered, if any of the following blocks of information are unknown at the time when the code spec is being created:

- The text of the SQL statement
- The number of columns
- The data types of columns
- References to database objects such as columns, indexes, sequences, tables, usernames and views

For example, dynamic SQL can be used to:

- Create a procedure that operates on a table whose name is not known until runtime
- Add a WHERE clause based on what fields are populated in a d/e form
- Create tables with user defined names in real time
- Run a complex query with a user-selectable sort order

DYNAMIC SQL STATEMENTS USING DBMS_SQL

All Oracle installations have a powerful package called DBMS_SQL installed that permits the construction and execution of Dynamic SQL statements. This package allows PL/SQL to execute SQL Data Definition Language statements [DDL] and Data Manipulation Language [DML] statements dynamically at run time within any PL/SQL block.

Tip

To use the **DBMS_SQL** package, the user requires appropriate database privileges.

Process Flow

Using the built-in DBMS_SQL package, dynamic SQL statements are processed as follows:

Open A Cursor And Maintain A Pointer To It

To construct and execute a dynamic SQL statement using the DBMS_SQL package, the very first step is to open a cursor.

Note

This cursor is completely different from the normal native PL/SQL cursor.

When such a cursor is opened, Oracle DB engine reserves memory (RAM) space for storing data retrieved from a table(s) into memory. After the cursor is opened, the Oracle Db engine returns an INTEGER handle, which can be used in all future calls to the DBMS_SQL package for access to the dynamic SQL statement. The integer pointer, called the Cursor Handle, points to the memory location that Oracle Db engine has reserved for table(s) data.

Syntax:

```
FUNCTION DBMS_SQL.OPEN_CURSOR RETURN <Integer>;
```

Parse SQL Statement For Syntax And Object Privileges

The dynamic SQL statement is broken down into words that can be categorized as:

- Commands such as SELECT, INSERT, UPDATE and DELETE
- Clauses such as WHERE, ORDER BY, GROUP BY, HAVING and so on
- Oracle objects such as user name, table names, column names and so on

The Oracle Db engine parses the SQL statement as follows:

Oracle commands/clauses and so on used in the SQL statement are compared with the ones available with Oracle and checked for syntax. This is done to ensure that the commands/clauses used in the SQL sentence are not only appropriate, but also placed at their correct positions.

Once the SQL verbs and their positions are identified as correct, the Oracle DB engine checks whether the Oracle objects are available within the database and the user who has fired the SQL sentence has valid permissions to use them.

Oracle associates the dynamic SQL statement with the cursor handle created earlier.

Syntax:

```
PROCEDURE DBMS_SQL.PARSE (<PointerTOCursor> IN Integer,
    <SQLStatement> IN Varchar2, <LanguageFlag> IN Integer);
```

Where,

- **PointerToCursor**
  Is the Integer value that points to the cursor (or memory area) reserved for the output of the SQL statement
- **SQLStatement**
  The SQL statement that must be parsed and associated with the cursor. This statement should not be terminated with a semicolon unless it is a PL/SQL block
- **LanguageFlag**
  A flag to indicate the language such as:
  - **DBMS_SQL.V6:** Use the Version 6 behavior when processing the statement
  - **DBMS_SQL.V7:** Use the Version 7 behavior
  - **DBMS_SQL.NATIVE:** Use the normal behavior for the database to which the program is connected

r storing data retrieved
returns an INTEGER
s to the dynamic SQL
cation that Oracle DB

egorized as:

he ones available with
ed in the SQL sentence

ne checks whether the
QL sentence has valid

the output of the SQL

tatement should not be

nt

which the program is

Bind Columns To The Cursor Columns

Since the SQL statement is dynamic, all the information about the SQL statement is not known at compile time. Hence, appropriate values must be passed to the SQL statement at runtime so that the sentence is complete.

Binding allows placing placeholders in the SQL statement and then explicitly binding or associate a value with that placeholder just prior executing the SQL statement.

Syntax:

```
PROCEDURE <BindVariable> (<CursorHandle> IN Integer ,
<VariableName> IN Varchar2, <Value> IN Integer);
PROCEDURE <BindVariable> (<CursorHandle> IN Integer ,
<VariableName> IN Varchar2, <Value> IN Date);
PROCEDURE <BindVariable> (<CursorHandle> IN Integer ,
<VariableName> IN Varchar2, <Value> IN Varchar2);
```

Where,

- **CursorHandle**
  Is the handle (or Integer value pointer) to the cursor originally returned by a call to OPEN_CURSOR
- **VariableName**
  Is the name of the host variable included in the SQL statement that has to be PARSED
- **Value**
  Is the value that must be bound to the host variable

Define Variable To Fetch Data From The Cursor Variables

For a SELECT SQL statement, the Oracle Db engine retrieves data from table(s) on the hard disk and stores this in the opened cursor.

Oracle then retrieves one row at a time from the cursor. A set of variables must be defined and mapped to the cursor columns before this data is fetched from the cursor.

Defining variables sets up a correspondence between the expressions in the list of the SQL statement and local PL/SQL variables receiving the values when a row is fetched.

The column must be defined using DEFINE_COLUMN. The defined column is then passed to the COLUMN_VALUE along with the cursor handle and the cursor column position.

When **DBMS_SQL.PARSE** is used to process a SELECT statement to retrieve values from database table columns to local variables, the columns or expressions in the SELECT list must be associated with the local variables. **DEFINE_COLUMN** is used for the purpose.

DEFINE_COLUMN is called **after** the call to **PARSE**. After the execution of the SELECT sentence **COLUMN_VALUE** is used to grab a column value from the SELECT list and pass it to appropriate local variables.

Syntax [Overloaded]:

```
PROCEDURE <DefineColumn> (<CursorHandle> IN Integer,
<Position> IN Integer, <Column> IN Date);
PROCEDURE <DefineColumn> (<CursorHandle> IN Integer,
<Position> IN Integer, <Column> IN Number);
PROCEDURE <DefineColumn> (<CursorHandle> IN Integer,
<Position> IN Varchar2, <Column> IN Integer,
<ColumnSize> IN Integer);
```

Where,

- **CursorHandle**
  Is the handle (or Integer value pointer) to the cursor originally returned by a call to OPEN_Cursor
- **Position**
  Is the relative position of the column in the SELECT list
- **Column**
  Is a local variable or expression whose data type determines the data type of the column being defined
- **ColumnSize**
  Is the size of the specified column

Execute Query To Fetch Data From Tables Into Cursor Columns

Once the required columns are defined, the SQL query must be executed to retrieve data from underlying tables.

Syntax:

```
FUNCTION EXECUTE (<CursorHandle IN INTEGER>) RETURN INTEGER;
```

Where,

- **CursorHandle**
  Is the Integer value pointer to the cursor

This function returns the number of rows processed by the SQL statement if that statement is an UPDATE, INSERT or DELETE. For all other SQL [queries and DDL] and PL/SQL statements, the value returned by EXECUTE is undefined and should be ignored.

Fetch Cursor Column Values For A Specific Row Into Memory Variables

Once the query is executed and data is available in the cursor, cursor column values must be assigned to memory variables.

Syntax:

```
FUNCTION FETCH_ROWS (<CursorHandle> IN <INTEGER>)
RETURN <NUMBER>;
```

Where,

- **CursorHandle**
  Is the Integer value pointer to the cursor

FETCH_ROWS returns 0 when there are no more rows to fetch. The **FETCH_ROWS** function can therefore be used like the **%FOUND** [or **%NOTFOUND**] attribute is used in normal cursors.

Retrieve Values For SQL Execution

This is equivalent to the **INTO** clause of an implicit SELECT statement in PL/SQL. This is done using the COLUMN_VALUE procedure.

Syntax[Overloaded]:

```
PROCEDURE <ColumnValue>(<CursorHandle> IN Integer,
   <Position> IN Integer, <Value> OUT Date [,
   <ColumnError> OUT Number] [, <ActualLength> OUT Integer] );
PROCEDURE <ColumnValue>(<CursorHandle> IN Integer,
   <Position> IN Integer, <Value> OUT Number [,
   <ColumnError> OUT Number] [, <ActualLength> OUT Integer] );
PROCEDURE <ColumnValue>(<CursorHandle> IN Integer,
   <Position> IN Integer, <Value> OUT Varchar2 [,
   <ColumnError> OUT Number] [, <ActualLength> OUT Integer] );
PROCEDURE <ColumnValue>(<CursorHandle> IN Integer,
   <Position> IN Integer, <Value> OUT Raw [,
   <ColumnError> OUT Number] [, <ActualLength> OUT Integer] );
PROCEDURE <ColumnValue>(<CursorHandle> IN Integer,
   <Position> IN Integer, <Value> OUT RowID [,
   <ColumnError> OUT Number] [, <ActualLength> OUT Integer] );
```

Where,

- **CursorHandle**
  Is the handle or pointer to the cursor originally returned by a call to the OPEN_CURSOR
- **Position**
  Is the relative position of the column in the SELECT list
- **Value**
  Is a local variable that will receive the outgoing value
- **ColumnError**
  Returns an error code for the specified value [the value returned may be larger than the variable can handle, for instance]
- **ActualLength**
  Returns the actual length of the returned value before any truncation takes place. This could be due to a difference in the size between the retrieved value in the cursor and the declared size of the variable

A **ColumnValue** procedure is called after a row has been fetched to transfer the values from the SELECT list of the cursor into local variables.

Perform Required Processing

Once cursor column values are assigned to memory variables, they can be used for further data processing.

Close The Cursor

Once the required data retrieved from the table is processed, the cursor memory area is freed by closing the cursor.

Syntax:

```
PROCEDURE <CloseCursor> ( <CursorHandle> IN OUT Integer);
```

The **CloseCursor** procedure closes the specified cursor. All memory locations associated with the cursor released. Additionally, the **CursorHandle** is set to **NULL**.

Example

Write a procedure that:

- Accepts a table name and a column name
- Displays the number of values held in that table's column

Solution:

```
CREATE OR REPLACE PROCEDURE CountRecords(
varTableName IN Varchar2, varColumnName IN Varchar2)
IS
/ * Declaring a handle to the Dynamic SQL cursor . * /
varCursor Number;
/ * Declaring a variable t hold the SQL query . * /
varSQL Varchar2(200);
/ * Declaring a variable to hold the return value from the
EXECUTE. */
varResult Number;
/ * Declaring a variable to hold the number of count . * /
varCount Number;

BEGIN
/ * Defining Dynamic SQL statement . * /
varSQL := 'SELECT COUNT (:ColName) ColumnCount
FROM ' || varTableName;
/ * Opening Cursor . * /
varCursor := DBMS_SQL.OPEN_CURSOR;
/ * Parsing SQL Statement . * /
DBMS_SQL.PARSE(varCursor, varSQL, DBMS_SQL.v7);
/ * Binding Data. * /
DBMS_SQL.BIND_VARIABLE (varCursor, 'ColName', varColumnName);
/ * Defining variables to fetch data from the cursor . * /
DBMS_SQL.DEFINE_COLUMN(varCursor, 1, varCount);
/ * Executing SQL statement . * /
varResult := DBMS_SQL.EXECUTE(varCursor);

/* Fetching and Processing . * /
LOOP
/ * Exit condition for the loop . * /
EXIT WHEN DBMS_SQL.FETCH_ROWS (varCursor) = 0;
/ * Transferring value from the SELECT column iist to the variable . * /
DBMS_SQL.COLUMN_VALUE(varcursor , 1, varCount);
END LOOP;
/ * Displaying the number of rows as a message . * /
DBMS_OUTPUT.PUT_LINE('Number of Rows in table named '
```

```
|| varTableName || ' is ' || varCount);
/* Closing Cursor . */
DBMS_SQL.CLOSE_CURSOR(varCursor);
END CountRecords;
/
```

Output:

```
Procedure created.
```

Test:

Test the procedure as:

```
SET SERVEROUTPUT ON
EXEC CountRecords('Employees','LastName');
```

Output:

```
Number of Rows in table named Employees is 11

PL/SQL procedure successfully completed.
```

Explanation:

Here, instead of using binding, concatenation can also be used to achieve the same results.

```
/* Defining Dynamic SQL statement . */
varSQL := 'SELECT COUNT (:ColName) ColumnCount FROM ' || varTableName;

/* Parsing SQL statement . */
DBMS_SQL.PARSE(varCursor, varSQL, DBMS_SQL.v7);
/* Binding Data . */
DBMS_SQL.BIND_VARIABLE (curPkey, 'ColName' , ColumnName);
DBMS_SQL.BIND_VARIABLE (curPkey, 'ColValue' , ColumnValue);
DBMS_SQL.BIND_VARIABLE (curPkey, 'TabName' , TableName);
```

Can be written using concatenation as:

```
/* Defining Dynamic SQL statement . */
varSQL := 'SELECT COUNT (' || varColumnName || ') ColumnCount
                FROM ' || varTableName;
/* Parsing SQL statement. */
DBMS_SQL.PARSE(varCursor , varSQL, DBMS_SQL.v7);
```

If concatenation is used then there is no binding at all and hence there is no need to call BIND_VARIABLE procedure.

Example:

Write a function that:

- Accepts:
  - Table name
  - Column name
  - Column value
- Determines if the column value is available in the column of that table
- If it is, then return 1 otherwise return 0

Solution:

```
CREATE OR REPLACE FUNCTION ChkAvailability (varTableName IN Varchar2,
varColumnName IN Varchar2, varColumnValue IN varchar2)
RETURN Number AS

/ * Declaring a handle to the Dynamic SQL cursor. * /
curChkAvl Integer;
/ * Declaring a variable to hold the return value from the EXECUTE. * /
EXECUTE_FEEDBACK Integer;

BEGIN
/ * Opening a cursor and storing the returned cursor ID . * /
curChkAvl := DBMS_SQL.OPEN_CURSOR;
/ * Parsing the SQL query with the columns in the SELECT list.
*/
DBMS_SQL.PRASE (curChkAvl, 'SELECT :ColumnValue
        FROM ' || varTableName || '
            WHERE :ColumnName = :ColumnValue', DBMS_SQL.V7);
/ * Binding columns to Cursor Columns . * /
   DBMS_SQL.BIND_VARIABLE (curChkAvl, 'ColumnName', varColumnName);
   DBMS_SQL.BIND_VARIABLE (curChkAvl, ColumnValue' , varColumnValue);
/ * Executing the SQL statement . * /
    EXECUTE_FEEDBACK := DBMS_SQL.EXECUTE(curChkAvl);

/ * Determining if the SQL query execution returned records . * /
  IF DBMS_SQL.FETCH_ROWS (curChkAvl) = 0 THEN
      RETURN 0;
ELSE
      RETURN 1;
END IF;
/ * Closing the Cursor . * /
    DBMS_SQL.CLOSE_CURSOR (curChkAvl);
END;
/
```

Output:

```
Function created.
```

Test:

Test the function ChkAvailability as:

```
SET SERVEROUTPUT ON
DECLARE
/ * Declaring a variable that holds the return value of the function . * /
   varAvailibility Number;

BEGIN
/ * Calling the function ChkAvailability that takes the table name and column name/value as parameters
and returns a number .
   * /
```

```
varAvailability := ChkAvailability ('&TABLENAME','&COLUMNNAME',
                   '&COLUMNVALUE');
/* Displaying an appropriate message to the user */
IF varAvailability = 1 THEN
   DBMS_OUTPUT.PUT_LINE ('The column value is available.');
ELSE
   DBMS_OUTPUT.PUT_LINE ('The column value is not available');
END IF;
END;
/
```

Output:

```
Enter value for tablename : Accounts
Enter value for columnname : AccountNo
old   7   :   varAvailability   :=   ChkAvailability   (   '   &TABLENAME',
'&COLUMNNAME',
new 7 : varAvailability := ChkAvailability ( 'Accounts', 'ACCOUNTNO',
Enter value for columnvalue:  101
old 8 :   ' &COLUMNVALUE') ;
new 8 : '101' ) ;
The column value is not available

PL/SQL procedure successfully completed.
```

Example:

Write a procedure that:

- Accepts an object name either as a complete object name or as wildcards
- Drops the object(s) matching the object name

Solution:

```
CREATE OR REPLACE PROCEDURE DestroyObject (varObjtype IN Varchar2,
   varObjName IN Varchar2) IS
/* Craeting a static cursor to retrieve user objects . */
 CURSOR OBJ_CUR IS
   SELECT OBJECT_NAME, OBJECT_TYPE FROM USER_OBJECTS
      WHERE OBJECT_NAME LIKE UPPER(varObjName)
         AND OBJECT_TYPE LIKE UPPER(varOBJType)
            ORDER BY OBJECT_NAME;
/* Declaring a handle to the Dynamic SQL cursor . */
curDrop Integer;

BEGIN
/* For each object in the cursor . */
FOR OBJ_REC IN OBJ_CUR
LOOP
/* Opening a cursor and returning cursor ID . */
  curDrop := DBMS_SQL.OPEN_CURSOR;
/* Parsing dynamic SQL command to drop the object . */
  DBMS_SQL.PARSE (curDrop, 'DROP' || OBJ_REC.OBJECT_TYPE || ' '
                          || OBJ_REC.OBJECT_NAME, DBMS_SQL.V7);
```

```
    /* Closing the cursor . */
        DBMS_SQL.CLOSE_CURSOR (curDrop);
    END LOOP;
END;
/
```

Output:

```
Procedure created.
```

Test:
Test the procedure as:
Check the existence of the view vwEmpDept as:

```
DESC vwEmpDept;
```

Output:

```
Name                                      Null ?    Type
----------------------------------------- --------- ------------------------------------
EMPLOYEENO                                NOT NUL   NUMBER (10)
FIRSTNAME                                           VARCHAR2 (25)
LASTNAME                                            VARCHAR2 (25)
SALARY                                              NUMBER (12 , 2)
DEPTNO                                    NOT NULL  NUMBER (10)
DEPARTMENTNAME                                      VARCHAR (25)
```

Drop the view:

```
EXEC DestroyObject('View', 'vwEmpDept');
```

Output:

```
PL/SQL procedure successfully completed.
```

Re-Check the existence of the view vwEmpDept:

```
DESC vwEmpDept;
```

Output:

```
ERROR :
ORA-04043 : object vwEmpDept does not exist
```

This procedure can also be passed wildcard as:

```
EXEC DestroyObject('View", '%Emp%');
```

Output:

```
PL/SQL procedure successfully completed.
```

This will drop all the views that have Emp in the view names.

Example:

Write a procedure that:

- Accepts a table name and a view name
- Create a view on all the columns of that table

Solution:

```
CREATE OR REPLACE PROCEDURE CreateView (varTableName IN Varchar2,
varViewName IN Varchar2) IS
    / * Declaring a handle to the Dynamic SQL cursor . * /
    curView Integer;
    BEGIN
        / * Opening a cursor and returning a cursor ID . * /
        curView := DBMS_SQL.OPEN_CURSOR;
        / * Parsing dynamic SQL command to create a view . * /
        DBMS_SQL.PARSE (curView, 'CREATE VIEW ' || varViewName || '
            AS SELECT * FROM || varTableName, DBMS_SQL.V7);
        / * Binding columns to Cursor Columns . * /
        DBMS_SQL.BIND_VARIABLE (curView, 'ViewName',varViewName);
        / * Closing the cursor . * /
        DBMS_SQL.CLOSE_CURSOR (curView);
END;
/
```

Output:

```
Procedure created.
```

Test:

Test the view as:

```
EXEC CreateView('Employees', 'vwEmp');
```

Output:

```
PL/SQL procedure successfully completed .
```

Query the view as:

```
SELECT EmployeeNo, FirstName, LastName, Designation, Salary FROM vwEmp;
```

Output:

EMPLOYEENO	FIRSTNAME	LASTNAME	DESIGNATION	SALARY
1	Sharanam	Shah	Technical Writing	82500
2	Vaishali	Shah	Database Management	55000
3	Keyur	Kapoor	Testing	22000
4	Amit	Haria	Programming	22000
5	Abhishek	Benegal	Sales	13334.2
6	Chetan	Dodhia	HR	13553.1
7	Narendra	Dodhia	Accounts	13578.4
8	Meenal	Doshi	Auditing	49500
9	Anil	Mayekar	Peon	2750
10	Chaitanya	Kapoor	Peon	1320
11	Amit	Paul	Peon	31207

11 rows selected.

DBMS_SQL CURSOR TO REF CURSOR AND VICE-VERSA

The DBMA_SQL package in Oracle Database 11g provides two new functions, TO_REFCURSOR, which converts the DBMS_SQL dynamic to a REF cursor and TO_CURSOR_NUMBER which converts a REFCURSOR to DBMS_SQL cursor.

This means both the cursors [i.e. DBMS_SQL and REFCURSOR] are friends and can talk to each other.

TO_REFCURSOR

TO_REFCURSOR converts a SQL cursor number to a weakly-typed variable of the PL/SQL data type REF CURSOR, which can be used in native dynamic SQL statements.

Before passing a SQL cursor number to the DBMS_SQL.TO_REFCURSOR function, ensure that cursor is OPENed, PARSEd, and EXECUTEd.

After a SQL cursor number is converted to a REF CUSOR variable, DBMS_SQL operations can access it only as the REF CURSOR variable, not as the SQL cursor number.

TO_CURSOR_NUMBER

TO_CURSOR_NUMBER converts a REF CURSOR variable [either strongly or weakly typed] to a SQL cursor number, which can be passed to DBMS_SQL susbprograms.

Before passing a REF CURSOR variable to the DBMS_SQL.TO_CURSOR_NUMBER function, it must be OPENed.

After a REF CURSOR variable is converted to a SQL cursor number, native dynamic SQL operations cannot access it.

Example:

Write a procedure that accepts:

- A REF Cursor as a parameter
- Displays the structure of that cursor

Solution:

```
CREATE OR REPLACE PROCEDURE DescStructure
        (varCursor SYS_REFCURSOR) IS
    / * Delcaring a REF Cursor . * /
      curRef SYS_REFCURSOR;
    / * Declaring a DBMS_SQL Cursor . * /
      curDyn Integer;
    / * Declaring a variable to hold the number of columns . * /
      varNoOfCols Integer;
    / * Decl;aring a record type . * /
      varDescTab DBMS_SQL.DESC_TAB;
    BEGIN
```

```
/* Assigning the cursor received as a parameter to a local REF Cursor . */
   curRef := varCursor;
/* Converting A REF Cursor TO DBMS_SQL Cursor. */
   curDyn := DBMS_SQL.TO_CURSOR_NUMBER(curRef);
/* Calling DESCRIBE_COLUMNS to populats the table with the
description of each column */
   DBMS_SQL.DESCRIBE_COLUMNS(curDyn, varNoOfCols, varDescTab);
/* Displaying the number of columns . */
   DBMS_OUTPUT.PUT_LINE('number of columns = ' || varNoOfCols);
/* Displaying the column description . */
   DBMS_OUTPUT.NEW_LINE;
   DBMS_OUTPUT.PUT_LINE('Columns');
   DBMS_OUTPUT.PUT_LINE('==========================================');
/* Traversing through the available columns . */
   FOR i IN 1 ..varNoOfCols
   LOOP
/* Retrieving the column description for each column in the loop. */
   DBMS_OUTPUT.PUT_LINE(varDescTab(i).col_name);
END LOOP;

/* Closing the cursor . */
   DBMS_SQL.CLOSE_CURSOR(curDyn);

/* Handling Exceptions . */
   EXECPTION
      WHEN OTHERS THEN
         IF DBMS_SQL.IS_OPEN(curDyn) THEN
            DBMS_SQL.CLOSE_CURSOR(curDyn);
         END IF;
      RAISE;
END DescStructure;
```

Output:

```
Procedure created.
```

Test:

Test the procedure as:

```
SET SERVEROUTPUT ON
DECLARE
    curTemp SYS_REFCURSOR;
BEGIN
   OPEN curTemp FOR SELECT * FROM Employess;
   DescStructure(curTemp);
END;
```

Output:

```
number of columns = 15
Columns
------------------------------
EMPLOYEENO
FIRSTNAME
LASTNAME
DATEOFBIRTH
GENDER
MARITALSTATUS
ADDRESS
CONTACTNO
SALARY
DESIGNATION
MANAGERNO
DEPTNO
ABOUTME
USERNAME
PASSWORD

PL/SQL procedure successfully completed.
```

SELF REVIEW QUESTIONS

FILL IN THE BLANKS

1. The _____ clause is another section of the select statement.
2. The _____ clause imposes a condition on the GROUP BY clause
3. A _____ is a form of an SQL statement that appears inside another SQL statement.
4. A subquery is also termed as _____ query.
5. The concept of joining multiple tables is called _____.
6. The _____ clause merges the output of two or more queries into a single set of rows or columns.
7. Multiple queries can be put together and their output combined using the _____ clause.

TRUE OR FALSE

8. The HAVING CLAUSE is an optional clause which tells Oracle to group rows based on distinct values that exist for specified columns.
9. The statement containing a subquery is called a parent statement.
10. Joining a table to itself is called Equi join.
11. If a select statement is defined as a subquery, the innermost select statement gets executed first.
12. In the union clause multiple queries can be put together but their outputs cannot be combined.
13. Unions can be used in subqueries.
14. The Intersect clause outputs only rows produ
15. The Minus clause outputs the rows produc the second query.

HANDS ON EXERCISES

1. **Exercises on using Having and Group**
   a. Print the description and total qty sold fo
   b. Find the value of each product sold.
   c. Calculate the average qty sold for each
   d. Find out the total of all the billed order

2. **Exercises on Joins and Correlation:**
   a. Find out the products, which have bee
   b. Find out the products and their quanti
   c. List the ProductNo and description o
   d. Find the names of clients who have
   e. List the products and orders from cu
   f. Find the products and their quantiti
   g. Find the products and their quantiti

3. **Exercise on Sub-queries:**
   a. Find the ProductNo and descriptio
   b. List the customer Name, Address 'O19001'.
   c. List the client names that have pl
   d. List if the product 'Lycra Top' h it was sold.
   e. List the names of clients who h

13. Unions can be used in subqueries.
14. The Intersect clause outputs only rows produced by both the queries intersected.
15. The Minus clause outputs the rows produced by the first query, before filtering the rows retrieved by the second query.

HANDS ON EXERCISES

1. **Exercises on using Having and Group By Clauses:**
a. Print the description and total qty sold for each product.
b. Find the value of each product sold.
c. Calculate the average qty sold for each client that has a maximum order value of 15000.00.
d. Find out the total of all the billed orders for the month of June.

2. **Exercises on Joins and Correlation:**
a. Find out the products, which have been sold to 'Ivan Bayross'.
b. Find out the products and their quantities that will have to be delivered in the current month.
c. List the ProductNo and description of constantly sold (i.e. rapidly moving) products.
d. Find the names of clients who have purchased 'Trousers'.
e. List the products and orders from customers who have ordered less than 5 units of 'Pull Overs'.
f. Find the products and their quantities for the orders placed by 'Ivan Bayross' and 'Mamta Muzumdar'.
g. Find the products and their quantities for the orders placed by ClientNo 'C00001' and 'C00002'

3. **Exercise on Sub-queries:**
a. Find the ProductNo and description of non-moving products i.e. products not being sold.
b. List the customer Name, Address1, Address2, City and PinCode for the client who has placed order no 'O19001'.
c. List the client names that have placed orders before the month of May'02.
d. List if the product 'Lycra Top' has been ordered by any client and print the Client_no, Name to whom it was sold.
e. List the names of clients who have placed orders worth Rs. 10000 or more.