My sincere **THANKS** go to:

- My publisher Mr. Manish Jain. He has brought enormous changes in my life. This is a debt of gratitude I will never be able to pay in full.

- Mr. Hansel Colaco who is in charge of quality control. Who personally took care that everything in the manuscript was rigidly bound to the specifications of our quality manual. You've done a really fine job.

- Mrs. Vaishali Shah who personally, laid out the manuscripts logic flow, tested the logic flow, grabbed the screens and structured much of the material. Vaishali without your tender, loving, care this manuscript would never have come together as it did. Your attention to detail was superb, you've done a really terrific job.

- Ms. Shweta Kini who checked and double-checked all the hands on exercises code spec and the solutions that accompanied the exercises. Thanks for being meticulous in your checking.

- Mr. Sharanam Shah the Tech lead on this project, your attention to detail in ANSI SQL syntax and its binding to commercial applications was excellent, you've done a really splendid job.

- The many SQL programmers who read this material, without you all I would not be an author. I welcome both your brickbats and bouquets. You can contact me via BpB, New Delhi else you could check out my web site appropriately named ivanbayross.com. Regretfully it's forever in a development stage.

- Finally, my wife Cynthia who has always encouraged me whenever I thought that I'd never get a manuscript ready for publishing. You have always helped to keep my feet firmly on the ground, with you I am truly blessed.

Ivan N. Bayross

TABLE OF CONTENTS

The Complete Entity Relationships For Fixed Deposit Creation

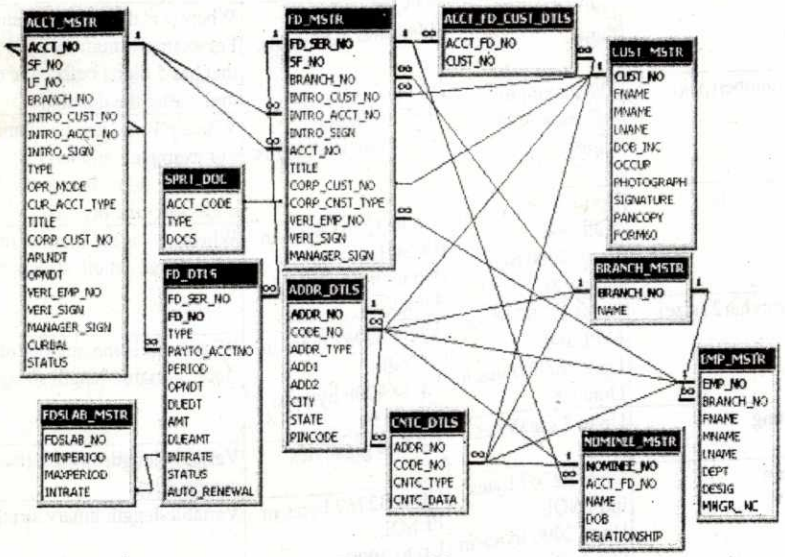

Diagram 6.9

Note

Refer to the file **Chap06_TestRecords.pdf**, (located under **More_Chaps** directory within the accompanying CD-ROM,) for the **test data** bound to tables of the **Retail Banking**.

SECTION III: Structured Query Language (SQL)

7. INTERACTIVE SQL PART - I

TABLE FUNDAMENTALS

A **table** is database object that holds user data. The simplest analogy is to think of a table as a spreadsheet. The cells of the spreadsheet equate to the columns of a table having a specific **data type** associated with them. If the spreadsheet cell has a number data type associated with it, then storing letters (**i.e.** characters) in the same cell is not allowed. The same logic is applied to a table's column. Each column of the table will have a specific data type bound to it. Oracle ensures that only data, which is identical to the data type of the column, will be stored within the column.

Oracle Data Types

Basic Data Types

Data types come in several forms and sizes, allowing the programmer to create tables suited to the scope of the project. The decisions made in choosing proper data types greatly influence the performance of a database, so it is wise to have a detailed understanding of these concepts.

Oracle is capable of many of the data types that even the novice programmer has probably already been exposed to. Refer to table 7.1 for some of the more commonly used include:

Data Type	Description
CHAR(size)	This data type is used to store character strings values of fixed length. The size in brackets determines the number of characters the cell can hold. The maximum number of characters (i.e. the size) this data type can hold is 255 characters. The data held is right-padded with spaces to whatever length specified. For example: In case of **Name CHAR(60)**, if the data held in the variable Name is only 20 characters in length, then the entry will be padded with 40 characters worth of spaces. These spaces will be removed when the value is retrieved though. These entries will be sorted and compared by MySQL in case-insensitive fashions unless the BINARY keyword is associated with it. The BINARY attribute means that column values are sorted and compared in case-sensitive fashion using the underlying character code values rather then a lexical ordering. BINARY doesn't affect how the column is stored or retrieved.
VARCHAR (size) / VARCHAR 2(size)	This data type is used to store variable length alphanumeric data. It is a more flexible form of the CHAR data type. The maximum this data type can hold upto 4000 characters. One difference between this data type and the CHAR data type is ORACLE compares VARCHAR values using non-padded comparison semantics i.e. the inserted values will not be padded with spaces. It also represents data of type String, yet stores this data in variable length format. VARCHAR can hold **1** to **255** characters. VARCHAR is usually a wiser choice than CHAR, due to it's variable length format characteristic. But, keep in mind, that CHAR is much faster than VARCHAR, sometimes up to **50%**.

Table 7.1

Data Type	Description
DATE	This data type is used to represent date and time. The standard format is DD-MON-YY as in 21-JUN-04. To enter dates other than the standard format, use the appropriate functions. DateTime stores date in the 24-hour format. By default, the time in a date field is 12:00:00 am, if no time portion is specified. The default date for a date field is the first day of the current month. Valid dates range from January 1, 4712 B.C. to December 31, 4712 A.D.
NUMBER (P, S)	The NUMBER data type is used to store numbers (fixed or floating point). Numbers of virtually any magnitude maybe stored up to 38 digits of precision. Valid values are 0, and positive and negative numbers with magnitude 1.0E-130 to 9.9...E125. Numbers may be expressed in two ways: first, with the numbers 0 to 9, the signs + and -, and a decimal point (.); second, in scientific notation, such as, 1.85E3 for 1850. The precision (P), determines the maximum length of the data, whereas the scale (S), determines the number of places to the right of the decimal. If scale is omitted then the default is zero. If precision is omitted, values are stored with their original precision upto the maximum of 38 digits.
LONG	This data type is used to store variable length character strings containing upto 2 GB. LONG data can be used to store arrays of binary data in ASCII format. Only one LONG value can be defined per table. LONG values cannot be used in subqueries, functions, expressions, where clauses or indexes and the normal character functions such as SUBSTR cannot be applied to LONG values. A table containing a LONG value cannot be clustered.
RAW/ LONG RAW	The RAW /LONG RAW data types are used to store binary data, such as digitized picture or image. Data loaded into columns of these data types are stored without any further conversion. RAW data type can have a maximum length of 255 bytes. LONG RAW data type can contain up to 2 GB. Values stored in columns having LONG RAW data type cannot be indexed.

Table 7.1 (Continued)

Comparison Between Oracle 8i/9i For Various Oracle Data Types

Data Type	Oracle 8i	Oracle 9i	Explanation
dec(p, s)	The maximum precision is 38 digits.	The maximum precision is 38 digits.	Where *p* is the precision and *s* is the scale. For example, dec(3,1) is a number that has 2 digits before the decimal and 1 digit after the decimal.
decimal(p, s)	The maximum precision is 38 digits.	The maximum precision is 38 digits.	Where *p* is the precision and *s* is the scale. For example, decimal(3,1) is a number that has 2 digits before the decimal and 1 digit after the decimal.
double precision			
float			
int			
integer			
real			
smallint			

Table 7.2

Data Type	Oracle 8i	Oracle 9i	Explanation
numeric(p, s)	The maximum precision is 38 digits.	The maximum precision is 38 digits.	Where *p* is the precision and *s* is the scale. For example, numeric(7,2) is a number that has 5 digits before the decimal and 2 digits after the decimal.
number(p, s)	The maximum precision is 38 digits.	The maximum precision is 38 digits.	Where *p* is the precision and *s* is the scale. For example, number(7,2) is a number that has 5 digits before the decimal and 2 digits after the decimal.
char (size)	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 8i.	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 9i.	Where *size* is the number of characters to store. Fixed-length strings. Space padded.
varchar2 (size)	Up to 32767 bytes in PLSQL. Up to 4000 bytes in Oracle 8i.	Up to 32767 bytes in PLSQL. Up to 4000 bytes in Oracle 9i.	Where *size* is the number of characters to store. Variable-length strings.
long	Up to 2 gigabytes.	Up to 2 gigabytes.	Variable-length strings. (backward compatible)
raw	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 8i.	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 9i.	Variable-length binary strings
long raw	Up to 2 gigabytes.	Up to 2 gigabytes.	Variable-length binary strings. (backward compatible)
date	A date between Jan 1, 4712 BC and Dec 31, 9999 AD.	A date between Jan 1, 4712 BC and Dec 31, 9999 AD.	
timestamp (*fractional seconds precision*)	Not supported in Oracle 8i.	*fractional seconds precision* must be a number between 0 and 9. (default is 6)	Includes year, month, day, hour, minute, and seconds. For example: timestamp(6)
timestamp (*fractional seconds precision*) with time zone	Not supported in Oracle 8i.	*fractional seconds precision* must be a number between 0 and 9. (default is 6)	Includes year, month, day, hour, minute, and seconds; with a time zone displacement value. For example: timestamp(5) with time zone
timestamp (*fractional seconds precision*) with local time zone	Not supported in Oracle 8i.	*fractional seconds precision* must be a number between 0 and 9. (default is 6)	Includes year, month, day, hour, minute, and seconds; with a time zone expressed as the session time zone. For example: timestamp(4) with local time zone
interval year (*year precision*) to month	Not supported in Oracle 8i.	*year precision* must be a number between 0 and 9. (default is 2)	Time period stored in years and months. For example: interval year(4) to month
urowid [size]	Up to 2000 bytes.	Up to 2000 bytes.	Universal rowid. Where *size* is optional.

Table 7.2 (Continued)

Data Type	Oracle 8i	Oracle 9i	Explanation
interval day (*day precision*) to second (*fractional seconds precision*)	Not supported in Oracle 8i.	*day precision* must be a number between 0 and 9. (default is 2) *fractional seconds precision* must be a number between 0 and 9. (default is 6)	Time period stored in days, hours, minutes, and seconds. For example: interval day(2) to second(6)
rowid	The format of the rowid is: BBBBBBB.RRRR.FFFFF Where BBBBBBB is the block in the database file; RRRR is the row in the block; FFFFF is the database file.	The format of the rowid is: BBBBBBB.RRRR.FFFFF Where BBBBBBB is the block in the database file; RRRR is the row in the block; FFFFF is the database file.	Fixed-length binary data. Every record in the database has a physical address or rowid.
boolean	Valid in PLSQL, but this datatype does not exist in Oracle 8i.	Valid in PLSQL, but this datatype does not exist in Oracle 9i.	
nchar (size)	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 8i	Up to 32767 bytes in PLSQL. Up to 2000 bytes in Oracle 9i.	Where *size* is the number of characters to store. Fixed-length NLS string
nvarchar2 (size)	Up to 32767 bytes in PLSQL. Up to 4000 bytes in Oracle 8i.	Up to 32767 bytes in PLSQL. Up to 4000 bytes in Oracle 9i.	Where *size* is the number of characters to store. Variable-length NLS string
bfile	Up to 4 gigabytes.	Up to 4 gigabytes.	File locators that point to a read-only binary object outside of the database
blob	U p to 4 gigabytes.	Up to 4 gigabytes.	LOB locators that point to a large binary object within the database
clob	Up to 4 gigabytes.	Up to 4 gigabytes.	LOB locators that point to a large character object within the database
nclob	Up to 4 gigabytes.	Up to 4 gigabytes.	LOB locators that point to a large NLS character object within the database

Table 7.2 (Continued)

Prior using a table to store user data it needs to be created. Table creation is done using the **Create Table** syntax. When Oracle creates a table in response to a create table command, it stores table structure information within its Data Dictionary.

The CREATE TABLE Command

The **CREATE TABLE** command defines each column of the table uniquely. Each column has a minimum of three attributes, a name, datatype and size (**i.e. column width**). Each table column definition is a single clause in the create table syntax. Each table column definition is separated from the other by a comma. Finally, the SQL statement is terminated with a semi colon.

Rules For Creating Tables

1. A name can have maximum upto 30 characters
2. Alphabets from A-Z, a-z and numbers from 0-9 are allowed
3. A name should begin with an alphabet
4. The use of the special character like _ is allowed and also recommended. (Special characters like $, # are allowed **only in Oracle**).
5. SQL reserved words **not** allowed. For Example: create, select, and so on.

Syntax:

```
CREATE TABLE <TableName>
    (<ColumnName1> <DataType>(<size>), <ColumnName2> <DataType>(<Size>));
```

Note

Each column must have a datatype. The column should either be defined as null or not null and if this value is left blank, the database assumes "null" as the default.

A Brief Checklist When Creating Tables

The following provides a small checklist for the issues that need to be considered before creating a table:

- What are the attributes of the rows to be stored?
- What are the data types of the attributes?
- Should varchar2 be used instead of char?
- Which columns should be used to build the primary key?
- Which columns do (not) allow null values? Which columns do / do not, allow duplicates?
- Are there default values for certain columns that **also** allow null values?

Example 1:
Create the **BRANCH_MSTR** table as shown in the **Chapter 6** along with the structure for other table belonging to the **Bank System**.

```
CREATE TABLE "DBA_BANKSYS"."BRANCH_MSTR"(
    "BRANCH_NO" VARCHAR2(10),
    "NAME" VARCHAR2(25));
```

Output:

```
Table created.
```

Note

All table columns belong to a **single record**. Therefore all the table column definitions are enclosed within parenthesis.

Inserting Data Into Tables

Once a table is created, the most natural thing to do is load this table with data to be manipulated later.

When inserting a single row of data into the table, the insert operation:

- Creates a new row (empty) in the database table
- Loads the values passed (by the SQL insert) into the columns specified

Syntax:

```
INSERT INTO <tablename> (<columnname1>, <columnname2>)
    VALUES (<expression>, <expression2>);
```

Example 2:
Insert the values into the BRANCH_MSTR table (For values refer to 6th chapter under Test Records)

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B1', 'Vile Parle (HO)');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B2', 'Andheri');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B3', 'Churchgate');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B4', 'Sion');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B5', 'Borivali');
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B6', 'Matunga');
```

Output for each of the above INSERT INTO statements:

```
1 row created.
```

Tip

Character expressions placed within the **INSERT INTO** statement must be enclosed in **single quotes** (').

In the **INSERT INTO** SQL sentence, table columns and values have a one to one relationship, (**i.e.** the first value described is inserted into the first column, and the second value described is inserted into the second column and so on).

Hence, in an **INSERT INTO** SQL sentence if there are exactly the same numbers of values as there are columns and the values are sequenced in exactly in accordance with the data type of the table columns, there is **no need** to indicate the column names.

However, if there are less values being described than there are columns in the table then it is **mandatory** to indicate **both** the table **column name** and its corresponding **value** in the **INSERT INTO** SQL sentence.

In the absence of mapping a table column name to a value in the **INSERT INTO** SQL sentence, the Oracle engine will not know which columns to insert the data into. This will generally cause a loss of data integrity. Then the data held within the table will be largely useless.

Note

Refer to the file **Chap07_Adtn.pdf**, for the **INSERT INTO** statement belonging to the remaining tables as mentioned in **Chapter 6**. These statements are built on the test data mentioned in **Chapter 6: Test Records For Retail Banking**.

VIEWING DATA IN THE TABLES

Once data has been inserted into a table, the next most logical operation would be to view what has been inserted. The **SELECT** SQL verb is used to achieve this. The SELECT command is used to retrieve rows selected from one or more tables.

All Rows And All Columns

In order to view global table data the **syntax** is:

```
SELECT <ColumnName 1> TO <ColumnName N> FROM TableName;
```

Note

Here, **ColumnName1 to ColumnName N** represents table column names.

Syntax:

```
SELECT * FROM <TableName>;
```

Example 3:
Show all employee numbers, first name, middle name and last name who work in the bank.

SELECT EMP_NO, FNAME, MNAME, LNAME **FROM** EMP_MSTR;

Output:

EMP_NO	FNAME	MNAME	LNAME
E1	Ivan	Nelson	Bayross
E2	Amit		Desai
E3	Maya	Mahima	Joshi
E4	Peter	Iyer	Joseph
E5	Mandhar	Dilip	Dalvi
E6	Sonal	Abdul	Khan
E7	Anil	Ashutosh	Kambli
E8	Seema	P.	Apte
E9	Vikram	Vilas	Randive
E10	Anjali	Sameer	Pathak

10 rows selected.

Example 4:
Show all the details related to the Fixed Deposit Slab

SELECT * FROM FDSLAB_MSTR;

Output:

FDSLAB_NO	MINPERIOD	MAXPERIOD	INTRATE
1	1	30	5
2	31	92	5.5
3	93	183	6
4	184	365	6.5
5	366	731	7.5

Output: (Continuted)

```
6          732        1097       8.5
7          1098       1829        10
7 rows selected.
```

Tip

When data from all rows and columns from the table are to be viewed the syntax of the SELECT statement will be: SELECT * FROM <TableName>;

Oracle allows the use of the Meta character asterisk (*), this is expanded by Oracle to mean all rows and all columns in the table.

The Oracle Server parses and compiles the SQL query, executes it, and retrieves data from all rows/columns from the table.

Filtering Table Data

While viewing data from a table it is rare that **all the data** from the table will be required **each time**. Hence, SQL provides a method of filtering table data that is not required.

The ways of filtering table data are:

- Selected columns and all rows
- Selected rows and all columns
- Selected columns and selected rows

Selected Columns And All Rows

The retrieval of specific columns from a table can be done as shown below:

Syntax:

SELECT <ColumnName1>, <ColumnName2> FROM <TableName>;

Example 5:
Show the first name and the last name of the bank employees

SELECT FNAME, LNAME **FROM** EMP_MSTR;

Output:

```
FNAME      LNAME
Ivan       Bayross
Amit       Desai
Maya       Joshi
Peter      Joseph
Mandhar    Dalvi
Sonal      Khan
Anil       Kambli
Seema      Apte
Vikram     Randive
Anjali     Pathak
10 rows selected.
```


Selected Rows And All Columns

If information of a particular client is to be retrieved from a table, its retrieval must be based on a **specific condition**.

The SELECT statement used until now displayed all rows. This is because there was no condition set that informed Oracle about how to choose a specific set of rows (**or** a specific row) from any table. Oracle provides the option of using a **WHERE Clause** in an SQL query to apply a filter on the rows retrieved.

When a where clause is added to the SQL query, the Oracle engine compares each record in the table with the condition specified in the where clause. The Oracle engine displays only those records that satisfy the specified condition.

Syntax:

SELECT * FROM <TableName> WHERE <Condition>;

Here, **<Condition>** is always quantified as **<ColumnName = Value>**

Example 6:
Display the branch details of the branch named Vile Parle (HO)

SELECT * FROM BRANCH_MSTR **WHERE** NAME = 'Vile Parle (HO)';

Output:

```
BRANCH_NO      NAME
B1             Vile Parle (HO)
```

Note

When specifying a condition in the **where clause** all standard operators such as logical, arithmetic, predicates and so on, can be used.

Selected Columns And Selected Rows

To view a specific set of rows and columns from a table the syntax will be as follows:

Syntax:

SELECT <ColumnName1>, <ColumnName2> FROM <TableName>
WHERE <Condition>;

Example 7:
List the savings bank account numbers and the branch to which they belong.

SELECT ACCT_NO, BRANCH_NO **FROM** ACCT_MSTR **WHERE** TYPE = 'SB';

Output:

```
ACCT_NO     BRANCH_NO
SB1         B1
SB3         B1
SB5         B6
SB6         B4
SB8         B2
SB9         B4
7 rows selected.
```

ELIMINATING DUPLICATE ROWS WHEN USING A SELECT STATEMENT

A table could hold duplicate rows. In such a case, to view only unique rows the distinct clause can be used.

The DISTINCT clause allows removing duplicates from the result set. The DISTINCT clause can only be used with select statements.

The **DISTINCT** clause scans through the values of the column/s specified and displays only unique values from amongst them.

Syntax:

```
SELECT DISTINCT <ColumnName1>, <ColumnName2> FROM <TableName>;
```

The **SELECT DISTINCT *** SQL syntax scans through **entire rows**, and eliminates rows that have exactly the same contents in each column.

Syntax:

```
SELECT DISTINCT * FROM <TableName>;
```

Example 8:
Show different types of occupations of the bank customers by eliminating the repeated occupations

SELECT DISTINCT OCCUP **FROM** CUST_MSTR;

Output:

```
OCCUP
------------------------------
Business
Community Welfare
Executive
Information Technology
Retail Business
Self Employed
Service
7 rows selected.
```

First insert one more record in the table **BRANCH_MSTR** so as to see the output for the next query example.

INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) **VALUES**('B6', 'Matunga');

Example 9:
Show only unique branch details.

SELECT DISTINCT * FROM BRANCH_MSTR;

The following output shows the entry for B6 only once even though entered twice in the table.

Output:

```
BRANCH_NO     NAME
---------     --------------------
B1            Vile Parle (HO)
B2            Andheri
B3            Churchgate
B4            Sion
B5            Borivali
B6            Matunga
6 rows selected.
```

SORTING DATA IN A TABLE

Oracle allows data from a table to be viewed in a sorted order. The rows retrieved from the table will be sorted in either **ascending** or **descending** order depending on the condition specified in the **SELECT** sentence. The syntax for viewing data in a sorted order is as follows:

Syntax:

```
SELECT * FROM <TableName>
   ORDER BY <ColumnName1>, <ColumnName2> <[Sort Order]>;
```

The ORDER BY clause sorts the result set based on the columns specified. The ORDER BY clause can only be used in SELECT statements.

Example 10:
Show details of the branch according to the branch's name.

SELECT * FROM BRANCH_MSTR **ORDER BY** NAME;

Output:

```
BRANCH_NO  NAME
---------  --------------------
B2         Andheri
B5         Borivali
B3         Churchgate
B6         Matunga
B6         Matunga
B4         Sion
B1         Vile Parle (HO)
7 rows selected.
```

For viewing data in descending sorted order the word DESC must be mentioned after the column name and before the semi colon in the order by clause. In case there is no mention of the sort order, the Oracle engine sorts in ascending order by default.

Example 11:
Show the details of the branch according to the branch's name in descending order.

SELECT * FROM BRANCH_MSTR **ORDER BY** NAME **DESC;**

Output:

BRANCH_NO	NAME
B1	Vile Parle (HO)
B4	Sion
B6	Matunga
B6	Matunga
B3	Churchgate
B5	Borivali
B2	Andheri

7 rows selected.

CREATING A TABLE FROM A TABLE

Syntax:

```
CREATE TABLE <TableName> (<ColumnName>, < ColumnName>)
    AS SELECT <ColumnName>, <ColumnName> FROM <TableName>
```

Example 12:
Create a table named ACCT_DTLS having three fields i.e. ACCT_NO, BRANCH_NO and CURBAL from the source table named ACCT_MSTR and rename the field CURBAL to BALANCE.

CREATE TABLE ACCT_DTLS (ACCT_NO, BRANCH_NO, BALANCE)
AS SELECT ACCT_NO, BRANCH_NO, CURBAL **FROM** ACCT_MSTR;

Output:
Table created.

Note

If the Source Table **Acct_Mstr** was populated with records then the target table **Acct_Dtls** will also be populated with the same.

The **Source** table is the table identified in the **SELECT** section of this SQL sentence. The **Target** table is one identified in the **CREATE** section of this SQL sentence. This SQL sentence populates the Target table with data from the Source table.

To create a Target table without the records from the source table (i.e. create the structure only), the select statement must have a **WHERE clause**. The **WHERE clause** must specify a condition that **cannot** be satisfied.

This means the **SELECT** statement in the CREATE TABLE definition **will not retrieve** any rows from the source table, it will just retrieve the table structure thus the target table will be created empty.

Example 13:
Create a table named ACCT_DTLS having three fields i.e. ACCT_NO, BRANCH_NO and CURBAL from the source table named ACCT_MSTR and rename the field CURBAL to BALANCE. The table ACCT_DTLS should not be populated with any records.

CREATE TABLE ACCT_DTLS (ACCT_NO, BRANCH_NO, BALANCE)
AS SELECT ACCT_NO, BRANCH_NO, CURBAL **FROM** ACCT_MSTR **WHERE 1=2;**

Output:
Table created.

INSERTING DATA INTO A TABLE FROM ANOTHER TABLE

In addition to inserting data one row at a time into a table, it is quite possible to populate a table with data that already exists in another table. The syntax for doing so is as follows:

Syntax:

```
INSERT INTO <TableName>
    SELECT <ColumnName 1>, <ColumnName N> FROM <TableName>;
```

Example 14:
Insert data in the table ACCT_DTLS using the table ACCT_MSTR as a source of data.

INSERT INTO ACCT_DTLS **SELECT** ACCT_NO, BRANCH_NO, CurBal **FROM** ACCT_MSTR;

Output:
10 rows created.

Insertion Of A Data Set Into A Table From Another Table

Syntax:

```
INSERT INTO <TableName> SELECT <ColumnName 1>, <ColumnName N>
    FROM <TableName> WHERE <Condition>;
```

Example 15:
Insert only the savings bank accounts details in the target table ACCT_DTLS.

INSERT INTO ACCT_DTLS **SELECT** ACCT_NO, BRANCH_NO, CurBal **FROM** ACCT_MSTR
WHERE ACCT_NO **LIKE** 'SB%';

Output:
6 rows created.

DELETE OPERATIONS

The DELETE command deletes rows from the table that satisfies the condition provided by its where clause, **and** returns the number of records deleted.

Caution

If a DELETE statement without a WHERE clause is issued then, all rows are deleted.

The verb **DELETE** in SQL is used to remove either:

- All the rows from a table

OR

- A set of rows from a table

Removal Of All Rows

Syntax:

```
DELETE FROM <TableName>;
```

Example 16:
Empty the ACCT_DTLS table

```
DELETE FROM ACCT_DTLS;
```

Output:

```
16 rows deleted.
```

Removal Of Specific Row(s)

Syntax:

```
DELETE FROM <TableName> WHERE <Condition>;
```

Example 17:
Remove only the savings bank accounts details from the ACCT_DTLS table.

```
DELETE FROM ACCT_DTLS WHERE ACCT_NO LIKE 'SB%';
```

Output:

```
6 rows deleted.
```

Removal Of Specific Row(s) Based On The Data Held By The Other Table

Sometimes it is desired to delete records in one table based on values in another table. Since it is not possible to list more than one table in the FROM clause while performing a delete, the EXISTS clause can be used.

Example 18:
Remove the address details of the customer named **Ivan**.

```
DELETE FROM ADDR_DTLS WHERE EXISTS(SELECT FNAME FROM CUST_MSTR
    WHERE CUST_MSTR.CUST_NO = ADDR_DTLS.CODE_NO
        AND CUST_MSTR.FNAME = 'Ivan');
```


Output:

```
1 row deleted.
```

Explanation:
This will delete all records in the ADDR_DTLS table where there is a record in the CUST_MSTR table whose FNAME is Ivan, and the CUST_NO field belonging to the table CUST_MSTR is the same as the CODE_NO belonging to the table ADDR_DTLS.

UPDATING THE CONTENTS OF A TABLE

The **UPDATE** command is used to change or modify data values in a table.

The verb update in SQL is used to either update:

- All the rows from a table

OR

- A select set of rows from a table

Updating All Rows

The **UPDATE** statement updates columns in the existing table's rows with new values. The **SET** clause indicates which column data should be modified and the new values that they should hold. The **WHERE** clause, if given, specifies which rows should be updated. Otherwise, **all** table rows are updated.

Syntax:

```
UPDATE <TableName>
    SET <ColumnName1> = <Expression1>, <ColumnName2> = <Expression2>;
```

Example 19:
Update the address details by changing its city name to Bombay

```
UPDATE ADDR_DTLS SET City = 'Bombay';
```

Output:

```
44 rows updated.
```

Updating Records Conditionally

Syntax:

```
UPDATE <TableName>
    SET <ColumnName1> = <Expression1>, <ColumnName2> = <Expression2>
        WHERE <Condition>;
```

Example 20:
Update the branch details by changing the Vile Parle (HO) to head office.

```
UPDATE BRANCH_MSTR SET NAME = 'Head Office'
    WHERE NAME = 'Vile Parle (HO)';
```

Output:
```
1 row updated.
```

MODIFYING THE STRUCTURE OF TABLES

The structure of a table can be modified by using the **ALTER TABLE** command. **ALTER TABLE** allows **changing the structure of an existing table**. With **ALTER TABLE** it is possible to **add** or **delete** columns, **create** or **destroy** indexes, **change the data type** of existing columns, or **rename columns** or the **table** itself.

ALTER TABLE works by making a temporary copy of the original table. The alteration is performed on the copy, then the original table is deleted and the new one is renamed. While **ALTER TABLE** is executing, the original table is still readable by users of Oracle.

Updates and **writes** to the table are **stalled** until the new table is ready, and then are automatically redirected to the new table **without any failed updates**.

Note

To use **ALTER TABLE**, the **ALTER**, **INSERT**, and **CREATE** privileges for the table are required.

Adding New Columns

Syntax:

```
ALTER TABLE <TableName>
   ADD(<NewColumnName> <Datatype> (<Size>),
       <NewColumnName> <Datatype> (<Size>)...);
```

Example 21:
Enter a new field called City in the table BRANCH_MSTR.

```
ALTER TABLE BRANCH_MSTR ADD (CITY VARCHAR2(25));
```

Output:
```
Table altered.
```

Dropping A Column From A Table

Syntax:

```
ALTER TABLE <TableName> DROP COLUMN <ColumnName>;
```

Example 22:
Drop the column city from the BRANCH_MSTR table.

```
ALTER TABLE BRANCH_MSTR DROP COLUMN CITY;
```

Output:
```
Table altered.
```


Modifying Existing Columns

Syntax:

```
ALTER TABLE <TableName>
      MODIFY (<ColumnName> <NewDatatype>(<NewSize>));
```

Example 23:
Alter the BRANCH_MSTR table to allow the NAME field to hold maximum of 30 characters

```
ALTER TABLE BRANCH_MSTR MODIFY (NAME varchar2(30));
```

Output:
```
Table altered.
```

Restrictions on the ALTER TABLE

The following tasks **cannot** be performed when using the **ALTER TABLE** clause:

- Change the name of the table
- Change the name of the column
- Decrease the size of a column if table data exists

RENAMING TABLES

Oracle allows renaming of tables. The rename operation is done **atomically**, which means that **no other thread** can access any of the tables while the rename process is running.

Note

To rename a table the ALTER and DROP privileges on the original table, and the CREATE and INSERT privileges on the new table are required.

To rename a table, the syntax is

Syntax:

```
RENAME <TableName> TO <NewTableName>
```

Example 24:
Change the name of branches table to branch table

```
RENAME BRANCH_MSTR TO BRANCHES;
```

Output:
```
Table renamed.
```

TRUNCATING TABLES

TRUNCATE TABLE empties a table completely. Logically, this is equivalent to a **DELETE** statement that deletes all rows, but there are practical differences under some circumstances.

TRUNCATE TABLE differs from **DELETE** in the following ways:

- Truncate operations drop and re-create the table, which is much faster than deleting rows one by one
- Truncate operations are not transaction-safe (i.e. an error will occur if an active transaction or an active table lock exists)
- The number of deleted rows are not returned

Syntax:

```
TRUNCATE TABLE <TableName>;
```

Example 25:
Truncate the table BRANCH_MSTR

TRUNCATE TABLE BRANCH_MSTR;

Output:

```
Table truncated.
```

DESTROYING TABLES

Sometimes tables within a particular database become obsolete and need to be discarded. In such situation using the DROP TABLE statement with the table name can destroy a specific table.

Syntax:

```
DROP TABLE <TableName>;
```

Caution

If a table is dropped all records held within it are lost and cannot be recovered.

Example 26:
Remove the table BRANCH_MSTR along with the data held.

DROP TABLE BRANCH_MSTR;

Output:

```
Table dropped.
```

CREATING SYNONYMS

A synonym is an alternative name for objects such as tables, views, sequences, stored procedures, and other database objects.

Syntax:

```
CREATE [OR REPLACE]  [PUBLIC] SYNONYM [SCHEMA .]
   SYNONYM_NAME FOR [SCHEMA .]
   OBJECT_NAME [@ DBLINK];
```


In the syntax,

- The **OR** replace phrase allows to recreate the synonym (if it already exists) without having to issue a DROP synonym command.
- The **PUBLIC** phrase means that the synonym is a public synonym and is accessible to all users. Remember though that the user must first have the appropriate privileges to the object to use the synonym.
- The **SCHEMA** phrase is the appropriate schema. If this phrase is omitted, Oracle assumes that a reference is made to the user's own schema.
- The **OBJECT_NAME** phrase is the name of the object for which you are creating the synonym. It can be one of the following:
  - Table
  - Package
  - View
  - Materialized View
  - Sequence
  - Java Class Schema Object
  - Stored Procedure
  - User-Defined Object
  - Function
  - Synonym

Example 27:
Create a synonym to a table named EMP held by the user SCOTT.

CREATE PUBLIC SYNONYM EMPLOYEES **FOR** SCOTT.EMP;

Output:

```
Synonym created.
```

Explanation:
Now, users of other schemas can reference the table EMP, which is now called as EMPLOYEES without having to prefix the table name with the schema named SCOTT. For example:

SELECT * FROM EMPLOYEES;

Dropping Synonyms

Syntax:

```
DROP [PUBLIC] SYNONYM [SCHEMA.]SYNONYM_NAME [FORCE];
```

In the syntax,

- The **PUBLIC** phrase allows to drop a public synonym. If public is specified, then there is no need to specify a schema.
- The **FORCE** phrase will force Oracle to drop the synonym even if it has dependencies. It is probably not a good idea to use the force phrase as it can cause invalidation of Oracle objects

Example 28:
Drop the public synonym named EMPLOYEES

DROP PUBLIC SYNONYM EMPLOYEES;

Output:

```
Synonym dropped.
```

EXAMINING OBJECTS CREATED BY A USER

Finding Out The Table/s Created By A User

The command shown below is used to determine the tables to which a user has access. The tables created under the **currently selected** tablespace are displayed.

Example 29:

SELECT * FROM TAB;

Output:

```
TNAME                TABTYPE     CLUSTERID
-------------------- ----------- ----------
ACCT_FD_CUST_DTLS    TABLE
ACCT_MSTR            TABLE
ADDR_DTLS            TABLE
BRANCH_MSTR          TABLE
CNTC_DTLS            TABLE
CUST_MSTR            TABLE
EMP_MSTR             TABLE
FDSLAB_MSTR          TABLE
FD_DTLS              TABLE
FD_MSTR              TABLE
NOMINEE_MSTR         TABLE
SPRT_DOC             TABLE
TRANS_DTLS           TABLE
TRANS_MSTR           TABLE
14 rows selected.
```

Displaying The Table Structure

To display information about the columns defined in a table use the following syntax

Syntax:

DESCRIBE <TableName>;

This command displays the column names, the data types and the special attributes connected to the table.

Example 30:
Show the table structure of table BRANCH_MSTR

DESCRIBE BRANCH_MSTR;

Output:

```
Name          Null?  Type
------------- -------- ------------
BRANCH_NO             VARCHAR2(10)
NAME                  VARCHAR2(25)
```


SELF REVIEW QUESTIONS

FILL IN THE BLANKS

1. A _____ is a database object that holds user data.
2. Table creation is done using the _____ syntax.
3. Character expressions placed within the insert into statement must be enclosed in _____ quotes.
4. Oracle provides the option of using a _____ _____ in an SQL query to apply a filter on the rows retrieved.
5. The _____ _____ SQL syntax scans through the values of the column/s specified and displays only unique values from amongst them.
6. The SQL sentence populates the _____ table with data from the _____ table.
7. The name of the column cannot be changed using the _____ _____ clause.
8. The _____ command is used to change or modify data values in a table.
9. All table columns belong to a _____ _____.

TRUE OR FALSE

10. If a spreadsheet has a number data type associated with, then it can store characters as well.
11. Each table column definition is separated from the other by a colon.
12. All table columns belong to a single record.
13. In the insert into SQL sentence table columns and values have a one to many relationship.
14. The SELECT DISTINCT SQL syntax scans through entire rows, and eliminates rows that have exactly the same contents in each column.
15. When specifying a condition in the where clause only logical standard operators can be used.
16. Oracle allows data from a table to be viewed in a sorted order.
17. In order to view the data in descending sorted order the word 'desc' must be mentioned after the column name and before the semi colon in the order by clause.
18. The MODIFY command is used to change or modify data values in a table.
19. The name of the table cannot be changed using the ALTER TABLE clause.

HANDS ON EXERCISES

1. Create the tables described below:

Table Name: CLIENT_MASTER
Description: Used to store client information.

Column Name	Data Type	Size	Default	Attributes
CLIENTNO	Varchar2	6		
NAME	Varchar2	20		
ADDRESS1	Varchar2	30		
ADDRESS2	Varchar2	30		
CITY	Varchar2	15		
PINCODE	Number	8		
STATE	Varchar2	15		
BALDUE	Number	10,2		

Table Name: PRODUCT_MASTER
Description: Used to store product information.

Column Name	Data Type	Size	Default	Attributes
PRODUCTNO	Varchar2	6		
DESCRIPTION	Varchar2	15		
PROFITPERCENT	Number	4,2		
UNITMEASURE	Varchar2	10		
QTYONHAND	Number	8		
REORDERLVL	Number	8		
SELLPRICE	Number	8,2		
COSTPRICE	Number	8,2		

Table Name: SALESMAN_MASTER
Description: Used to store salesman information working for the company.

Column Name	Data Type	Size	Default	Attributes
SALESMANNO	Varchar2	6		
SALESMANNAME	Varchar2	20		
ADDRESS1	Varchar2	30		
ADDRESS2	Varchar2	30		
CITY	Varchar2	20		
PINCODE	Number	8		
STATE	Varchar2	20		
SALAMT	Number	8,2		
TGTTOGET	Number	6,2		
YTDSALES	Number	6,2		
REMARKS	Varchar2	60		

2. Insert the following data into their respective tables:

a) Data for **CLIENT_MASTER** table:

ClientNo	Name	City	Pincode	State	BalDue
C00001	Ivan Bayross	Mumbai	400054	Maharashtra	15000
C00002	Mamta Muzumdar	Madras	780001	Tamil Nadu	0
C00003	Chhaya Bankar	Mumbai	400057	Maharashtra	5000
C00004	Ashwini Joshi	Bangalore	560001	Karnataka	0
C00005	Hansel Colaco	Mumbai	400060	Maharashtra	2000
C00006	Deepak Sharma	Mangalore	560050	Karnataka	0

b) Data for **PRODUCT_MASTER** table:

ProductNo	Description	Profit Percent	Unit Measure	QtyOn Hand	ReorderLvl	SellPrice	CostPrice
P00001	T-Shirts	5	Piece	200	50	350	250
P0345	Shirts	6	Piece	150	50	500	350
P06734	Cotton Jeans	5	Piece	100	20	600	450
P07865	Jeans	5	Piece	100	20	750	500
P07868	Trousers	2	Piece	150	50	850	550
P07885	Pull Overs	2.5	Piece	80	30	700	450
P07965	Denim Shirts	4	Piece	100	40	350	250
P07975	Lycra Tops	5	Piece	70	30	300	175
P08865	Skirts	5	Piece	75	30	450	300

c) Data for **SALESMAN_MASTER** table:

SalesmanNo	Name	Address1	Address2	City	PinCode	State
S00001	Aman	A/14	Worli	Mumbai	400002	Maharashtra
S00002	Omkar	65	Nariman	Mumbai	400001	Maharashtra
S00003	Raj	P-7	Bandra	Mumbai	400032	Maharashtra
S00004	Ashish	A/5	Juhu	Mumbai	400044	Maharashtra

SalesmanNo	SalAmt	TgtToGet	YtdSales	Remarks
S00001	3000	100	50	Good
S00002	3000	200	100	Good
S00003	3000	200	100	Good
S00004	3500	200	150	Good

3. Exercise on retrieving records from a table
a. Find out the names of all the clients.
b. Retrieve the entire contents of the Client_Master table.
c. Retrieve the list of names, city and the sate of all the clients.
d. List the various products available from the Product_Master table.
e. List all the clients who are located in Mumbai.
f. Find the names of salesmen who have a salary equal to Rs.3000.

4. Exercise on updating records in a table
a. Change the city of ClientNo 'C00005' to 'Bangalore'.
b. Change the BalDue of ClientNo 'C00001' to Rs. 1000.
c. Change the cost price of 'Trousers' to Rs. 950.00.
d. Change the city of the salesman to Pune.

5. Exercise on deleting records in a table
a. Delete all salesmen from the Salesman_Master whose salaries are equal to Rs. 3500.
b. Delete all products from Product_Master where the quantity on hand is equal to 100.
c. Delete from Client_Master where the column state holds the value 'Tamil Nadu'.

6. Exercise on altering the table structure
a. Add a column called 'Telephone' of data type 'number' and size ='10' to the Client_Master table.
b. Change the size of SellPrice column in Product_Master to 10,2.

7. Exercise on deleting the table structure along with the data
a. Destroy the table Client_Master along with its data.

8. Exercise on renaming the table
a. Change the name of the Salesman_Master table to sman_mast.

Not NULL } column.
DEFault }

User - constraints.

8. INTERACTIVE SQL PART – II

DATA CONSTRAINTS

All businesses of the world run on business data being gathered, stored and analyzed. Business managers determine a set of business rules that must be applied to their data prior to it being stored in the database/table to ensure its integrity.

For instance, no employee in the sales department can have a salary of less than Rs.1000/-.

Such rules have to be enforced on data stored. Only data, which satisfies the conditions set, should be stored for future analysis. If the data gathered fails to satisfy the conditions set, it must be rejected. This ensures that the data stored in a table will be valid, and have integrity.

Business rules that are applied to data are completely **System dependent**. The rules applied to data gathered and processed by a **Savings bank system** will be very different, to the business rules applied to data gathered and processed by an **Inventory system**, which in turn will be very different, to the business rules applied to data gathered and processed by a **Personnel management system**.

Business rules, which are enforced on data being stored in a table, are called **Constraints**. Constraints, **super control** the data being entered into a table for permanent storage.

To understand the concept of data constraints, several tables will be created and different types of constraints will be applied to table columns **or** the table itself. The set of tables are described below. Appropriate examples of data constraints are bound to these tables.

Applying Data Constraints

Oracle permits data constraints to be attached to table columns via SQL syntax that checks data for integrity prior storage. Once data constraints are part of a table column construct, the Oracle database engine checks the data being entered into a table column against the data constraints. If the data passes this check, it is stored in the table column, else the data is rejected. Even if a single column of the record being entered into the table fails a constraint, the **entire record is rejected and not stored in the table**.

Both the **Create Table** and **Alter Table** SQL verbs can be used to write SQL sentences that attach constraints (i.e. Business / System rules) to a table column.

Caution

Until now tables created in this material have **not** had any data constraints attached to their table columns. Hence the tables have **not** been given any instructions to filter what is being stored in the table. This situation **can** and **does**, result in erroneous data being stored in the table.

Once a constraint is attached to a table column, any SQL **INSERT** or **UPDATE** statement automatically causes these constraints to be applied to data prior it is being inserted into the table column for storage.

Note

Oracle also permits applying data constraints at **Table level**. More on table level constraints later in this material.

TYPES OF DATA CONSTRAINTS

There are two types of data constraints that can be applied to data being inserted into a Oracle table. One type of constraint is called an **I/O** constraint (input / output). This data constraint determines the speed at which data can be inserted or extracted from a Oracle table. The other type of constraint is called a **business rule** constraint.

I/O Constraints

The input/output data constraints are further divided into **two** distinctly different constraints.

The PRIMARY KEY Constraint

A primary key is one or more column(s) in a table used to uniquely identify **each row** in the table. None of the fields that are part of the primary key can contain a null value. A table can have only one primary key. A **primary key column** in a table has special attributes:

- It defines the column, as a mandatory column (**i.e.** the column cannot be left blank). As the NOT NULL attribute is active
- The data held across the column MUST be UNIQUE

A single column primary key is called a **Simple** key. A multicolumn primary key is called a **Composite** primary key. The only function of a primary key in a table is to **uniquely identify a row**. When a record cannot be uniquely identified using a value in a simple key, a composite key must be defined. A primary key can be defined in either a **CREATE TABLE** statement or an **ALTER TABLE** statement.

For example, a **SALES_ORDER_DETAILS** table will hold multiple records that are sales orders. Each such sales order will have multiple products that have been ordered. Standard business rules do not allow multiple entries for the same product. However, multiple orders will definitely have multiple entries of the same product.

Under these circumstances, the only way to uniquely identify a row in the **SALES_ORDER_DETAILS** table is via a composite primary key, consisting of **ORDER_NO** and **PRODUCT_NO**. Thus the combination of order number and product number will uniquely identify a row.

Features of Primary key

1. Primary key is a column or a set of columns that uniquely identifies a row. Its main purpose is the **Record Uniqueness**
2. Primary key will not allow duplicate values
3. Primary key will also not allow null values
4. Primary key is not compulsory but it is recommended
5. Primary key helps to identify one record from another record and also helps in relating tables with one another
6. Primary key cannot be LONG or LONG RAW data type
7. Only one Primary key is allowed per table
8. Unique Index is created automatically if there is a Primary key
9. One table can combine upto 16 columns in a Composite Primary key

PRIMARY KEY Constraint Defined At Column Level

Syntax:

<ColumnName> <Datatype>(<Size>) PRIMARY KEY

Example 1:
Drop the CUST_MSTR table, if it already exists. Create a table CUST_MSTR such that the contents of the column CUST_NO is unique and not null.

```
DROP TABLE CUST_MSTR;
CREATE TABLE CUST_MSTR (
    "CUST_NO" VARCHAR2(10) PRIMARY KEY,
    "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DOB_INC" DATE,
    "OCCUP" VARCHAR2(25), "PHOTOGRAPH" VARCHAR2(25),
    "SIGNATURE"  VARCHAR2(25), "PANCOPY" VARCHAR2(1),
    "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

For testing purpose, execute the following **INSERT INTO** statement:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C1', 'Ivan', 'Nelson', 'Bayross', '25-JUN-1952', 'Self Employed', 'D:/ClntPht/C1.gif',
           'D:/ClntSgnt/C1.gif', 'Y', 'Y');
```

Output:

```
1 row created.
```

To verify whether the Primary Key Constraint is functional, reissue the same **INSERT INTO** statement. The result is the following error:

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-00001: unique constraint (DBA_BANKSYS.SYS_C003009) violated
```

PRIMARY KEY Constraint Defined At Table Level

Syntax:

PRIMARY KEY (<ColumnName>, <ColumnName>)

Example 2:
Drop the FD_MSTR table, if it already exists. Create a table FD_MSTR where there is a composite primary key mapped to the columns FD_SER_NO and CORP_CUST_NO.
Since this constraint spans across columns, it must be described at table level.

```
DROP TABLE FD_MSTR;
CREATE TABLE "DBA_BANKSYS"."FD_MSTR"(
    "FD_SER_NO" VARCHAR2(10), "SF_NO" VARCHAR2(10),
    "BRANCH_NO" VARCHAR2(10), "INTRO_CUST_NO" VARCHAR2(10),
    "INTRO_ACCT_NO" VARCHAR2(10), "INTRO_SIGN" VARCHAR2(1),
    "ACCT_NO" VARCHAR2(10), "TITLE" VARCHAR2(30),
    "CORP_CUST_NO" VARCHAR2(10), "CORP_CNST_TYPE" VARCHAR(4),
    "VERI_EMP_NO" VARCHAR2(10), "VERI_SIGN" VARCHAR2(1),
    "MANAGER_SIGN" VARCHAR2(1), PRIMARY KEY(FD_SER_NO, CORP_CUST_NO));
```

Output:
```
Table created.
```

For testing purpose, execute the following **INSERT INTO** statement:
```
INSERT INTO FD_MSTR (FD_SER_NO, SF_NO, BRANCH_NO, ACCT_NO, TITLE, CORP_CUST_NO,
                     CORP_CNST_TYPE, INTRO_CUST_NO, INTRO_ACCT_NO, INTRO_SIGN,
                     VERI_EMP_NO, VERI_SIGN, MANAGER_SIGN)
    VALUES ('FS1', 'SF-0011', 'B1', 'CA2', 'Uttam Stores', 'O11', '1C', null, null, 'N', 'E1', 'Y', 'Y');
```

Output:
```
1 row created.
```

To verify whether the Composite Primary Key Constraint is functional, reissue the same **INSERT INTO** statement. The result is the following error:

Output:
```
INSERT INTO FD_MSTR (FD_SER_NO, SF_NO, BRANCH_NO, ACCT_NO, TITLE,
CORP_CUST_NO, CORP_CNST_TYPE, INTR
*
ERROR at line 1:
ORA-00001: unique constraint (DBA_BANKSYS.SYS_C003010) violated
```

Now, simply modify the INSERT INTO statement as show below, to allow the record to pass the composite primary key constraint:
```
INSERT INTO FD_MSTR (FD_SER_NO, SF_NO, BRANCH_NO, ACCT_NO, TITLE, CORP_CUST_NO,
                     CORP_CNST_TYPE, INTRO_CUST_NO, INTRO_ACCT_NO, INTRO_SIGN,
                     VERI_EMP_NO, VERI_SIGN, MANAGER_SIGN)
    VALUES ('FS2', 'SF-0012', 'B1', 'CA4', 'Sun"s Pvt. Ltd.', 'C12', '4C', null, null, 'N', 'E1', 'Y', 'Y');
```

Output:
```
1 row created.
```

The Foreign Key (Self Reference) Constraint

Foreign keys represent relationships between tables. A foreign key is a column (or a group of columns) whose values are derived from the **primary key** or **unique key** of some other table.

The table in which the foreign key is defined is called a **Foreign table** or **Detail table**. The table that defines the **primary** or **unique** key and is referenced by the **foreign key** is called the **Primary table** or **Master table**. A Foreign key can be defined in either a CREATE TABLE statement or an ALTER TABLE statement

The master table can be referenced in the foreign key definition by using the clause **REFERENCES TableName.ColumnName** when defining the foreign key, column attributes, in the detail table.

Features of Foreign Keys

1. Foreign key is a column(s) that references a column(s) of a table and it can be the same table also
2. Parent that is being referenced has to be unique or Primary key
3. Child may have duplicates and nulls but unless it is specified
4. Foreign key constraint can be specified on child but not on parent
5. Parent record can be delete provided no child record exist
6. Master table cannot be updated if child record exist

This constraint establishes a relationship between records (i.e. column data) across a Master and a Detail table. This relationship ensures:

- Records cannot be **inserted** into a **detail table** if corresponding records in the master table do not exist
- Records of the master **table** cannot be **deleted** if corresponding records in the detail table actually exist

Insert Or Update Operation In The Foreign Key Table

The existence of a foreign key implies that the table with the foreign key is **related** to the master table from which the foreign key is derived. A foreign key must have a corresponding primary key or unique key value in the master table.

For example a personnel information system includes two tables (i.e. department and employee). An employee cannot belong to a department that does not exist. Thus the department number specified in the employee table must be present in the department table.

Delete Operation On The Primary Key Table

Oracle **displays an error message** when a record in the master table is deleted and corresponding records exists in a detail table and **prevents the delete operation** from going through.

Note

The default behavior of the foreign key can be changed, by using the **ON DELETE CASCADE** option. When the **ON DELETE CASCADE** option is specified in the foreign key definition, if a record is deleted in the master table, all corresponding records in the detail table along with the record in the master table will be deleted.

Principles of **Foreign Key/References** constraint:

- Rejects an **INSERT** or **UPDATE** of a value, if a corresponding value does not currently exist in the master key table
- If the **ON DELETE CASCADE** option is set, a **DELETE** operation in the master table will trigger a **DELETE** operation for corresponding records **in all** detail tables
- If the **ON DELETE SET NULL** option is set, a **DELETE** operation in the master table will set the value held by the foreign key of the detail tables to null
- **Rejects a DELETE** from the Master table if **corresponding records** in the DETAIL table **exist**
- Must reference a **PRIMARY KEY** or **UNIQUE** column(s) in primary table
- Requires that the **FOREIGN KEY** column(s) and the **CONSTRAINT** column(s) have **matching** data types
- Can reference the same table named in the **CREATE TABLE** statement

FOREIGN KEY Constraint Defined At The Column Level

Syntax:
```
<ColumnName> <DataType>(<Size>)
    REFERENCES <TableName> [(<ColumnName>)]
        [ON DELETE CASCADE]
```

Example 3:
Drop the table EMP_MSTR, if it already exists. Create a table EMP_MSTR with its primary as EMP_NO referencing the foreign key BRANCH_NO in the BRANCH_MSTR table.

```
DROP TABLE EMP_MSTR;
CREATE TABLE "DBA_BANKSYS"."EMP_MSTR"(
    "EMP_NO" VARCHAR2(10) PRIMARY KEY,
    "BRANCH_NO" VARCHAR2(10) REFERENCES BRANCH_MSTR,
    "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DEPT" VARCHAR2(30),
    "DESIG" VARCHAR2(30));
```

Output:
```
Table created.
```

The **REFERENCES** key word points to the table **BRANCH_MSTR**. The table **BRANCH_MSTR** has the column **BRANCH_NO** as its primary key column. Since no column is specified in the foreign key definition, Oracle applies an automatic (default) link to the primary key column i.e. **BRANCH_NO** of the table **BRANCH_MSTR**.

The foreign key definition is specified as

```
"BRANCH_NO" VARCHAR2(10) REFERENCES BRANCH_MSTR
```

FOREIGN KEY Constraint Defined At The Table Level

Syntax:

```
FOREIGN KEY (<ColumnName> [,<ColumnName>] )
    REFERENCES <TableName> [(<ColumnName>,<ColumnName>)
```

Example 4:
Drop the table ACCT_FD_CUST_DTLS, if it already exists. Create a table ACCT_FD_CUST_DTLS with CUST_NO as foreign key referencing column CUST_NO in the CUST_MSTR table

```
DROP TABLE ACCT_FD_CUST_DTLS;
CREATE TABLE "DBA_BANKSYS"."ACCT_FD_CUST_DTLS"(
    "ACCT_FD_NO" VARCHAR2(10), "CUST_NO" VARCHAR2(10),
    FOREIGN KEY (CUST_NO) REFERENCES CUST_MSTR(CUST_NO));
```

Output:
```
Table created.
```

FOREIGN KEY Constraint Defined With ON DELETE CASCADE

Example 5:
Drop the table FD_MSTR, if it already exists. Create a table FD_MSTR with its primary key as FD_SER_NO.

Drop the table FD_DTLS, if it already exists. Create a table FD_DTLS with its foreign key as FD_SER_NO with the ON DELETE CASCADE option. The foreign key is FD_SER_NO and is available as a primary key column named FD_SER_NO in the FD_MSTR table.

Insert some records into both the tables.

```
DROP TABLE FD_MSTR;
CREATE TABLE "DBA_BANKSYS"."FD_MSTR"(
        "FD_SER_NO" VARCHAR2(10) PRIMARY KEY,
        "SF_NO" VARCHAR2(10), "BRANCH_NO" VARCHAR2(10),
        "INTRO_CUST_NO" VARCHAR2(10), "INTRO_ACCT_NO" VARCHAR2(10),
        "INTRO_SIGN" VARCHAR2(1), "ACCT_NO" VARCHAR2(10),
        "TITLE" VARCHAR2(30), "CORP_CUST_NO" VARCHAR2(10),
        "CORP_CNST_TYPE" VARCHAR(4), "VERI_EMP_NO" VARCHAR2(10),
        "VERI_SIGN" VARCHAR2(1), "MANAGER_SIGN" VARCHAR2(1));
```

Output:
```
Table created.
```



```
DROP TABLE FD_DTLS;
CREATE TABLE "DBA_BANKSYS"."FD_DTLS"(
    "FD_SER_NO" VARCHAR2(10), "FD_NO" VARCHAR2(10),
    "TYPE" VARCHAR2(1), "PAYTO_ACCTNO" VARCHAR2(10),
    "PERIOD" NUMBER(5), "OPNDT" DATE,
    "DUEDT" DATE, "AMT" NUMBER(8,2),
    "DUEAMT" NUMBER(8,2), "INTRATE" NUMBER(3),
    "STATUS" VARCHAR2(1) DEFAULT 'A', "AUTO_RENEWAL" VARCHAR2(1) ,
    CONSTRAINT f_FDSerNoKey
        FOREIGN KEY (FD_SER_NO) REFERENCES FD_MSTR(FD_SER_NO)
        ON DELETE CASCADE);
```

Output:
```
Table created.
```

Now delete a record from the FD_MSTR table as:

```
DELETE FROM FD_MSTR WHERE FD_SER_NO = 'FS1';
```

Output:
```
1 row deleted.
```

Query the table FD_DTLS for records:

```
SELECT * FROM FD_DTLS;
```

Notice the deletion of the records belonging to **FS1**.

Explanation:
In this example, a primary key is created in the FD_MSTR table. It consists of only **one field** i.e. FD_SER_NO field. Then a foreign key is created in the FD_DTLS table that references the FD_MSTR table based on the contents of the FD_SER_NO field.

Because of the **cascade delete**, when a record in the FD_MSTR table is deleted, all records in the FD_DTLS table will also be deleted that have the same FD_SER_NO value.

FOREIGN KEY Constraint Defined With ON DELETE SET NULL:

A FOREIGN key with a **SET NULL ON DELETE** means that if a record in the parent table is deleted, then the corresponding records in the child table will have the foreign key fields set to **null**. The records in the child table **will not** be deleted.

A FOREIGN key with a **SET NULL ON DELETE** can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

Example 6:
Drop the table FD_MSTR, if it already exists. Create a table FD_MSTR with its primary key as FD_SER_NO.

Drop the table FD_DTLS, if it already exists. Create a table FD_DTLS with its foreign key as FD_SER_NO with the ON DELETE SET NULL option. The foreign key is FD_SER_NO and is available as a primary key column named FD_SER_NO in the FD_MSTR table.

Insert some records into both the tables.

```
DROP TABLE FD_MSTR;
```

```
CREATE TABLE "DBA_BANKSYS"."FD_MSTR"(
        "FD_SER_NO" VARCHAR2(10) PRIMARY KEY, "SF_NO" VARCHAR2(10),
        "BRANCH_NO" VARCHAR2(10), "INTRO_CUST_NO" VARCHAR2(10),
        "INTRO_ACCT_NO" VARCHAR2(10), "INTRO_SIGN" VARCHAR2(1),
        "ACCT_NO" VARCHAR2(10), "TITLE" VARCHAR2(30), "CORP_CUST_NO" VARCHAR2(10),
        "CORP_CNST_TYPE" VARCHAR(4), "VERI_EMP_NO" VARCHAR2(10),
        "VERI_SIGN" VARCHAR2(1), "MANAGER_SIGN" VARCHAR2(1));
```

Output:

```
Table created.
```

```
DROP TABLE FD_DTLS;
CREATE TABLE "DBA_BANKSYS"."FD_DTLS"(
    "FD_SER_NO" VARCHAR2(10), "FD_NO" VARCHAR2(10), "TYPE" VARCHAR2(1),
    "PAYTO_ACCTNO" VARCHAR2(10), "PERIOD" NUMBER(5), "OPNDT" DATE,
    "DUEDT" DATE, "AMT" NUMBER(8,2), "DUEAMT" NUMBER(8,2), "INTRATE" NUMBER(3),
    "STATUS" VARCHAR2(1) DEFAULT 'A', "AUTO_RENEWAL" VARCHAR2(1) ,
    CONSTRAINT f_FDSerNoKey
        FOREIGN KEY (FD_SER_NO) REFERENCES FD_MSTR(FD_SER_NO)
        ON DELETE SET NULL);
```

Output:

```
Table created.
```

Now delete a record from the FD_MSTR table as:

```
DELETE FROM FD_MSTR WHERE FD_SER_NO = 'FS1';
```

Output:

```
1 row deleted.
```

Query the table FD_DTLS for records:

```
SELECT * FROM FD_DTLS;
```

Notice the value held by the field FD_SER_NO.

Explanation:
In this example, a primary key is created in the FD_MSTR table. It consists of only one field i.e. FD_SER_NO field. Then a foreign key is created in the FD_DTLS table that references the FD_MSTR table based on the FD_SER_NO field.

Because of the **cascade set null**, when a record in the FD_MSTR table is deleted, all corresponding records in the FD_DTLS table will have the FD_SER_NO values set to **null**.

Assigning User Defined Names To Constraints

When constraints are defined, Oracle assigns a **unique name** to each constraint. The convention used by Oracle is

SYS_Cn

where **n** is a **numeric value** that makes the constraint name **unique**.

Constraints can be given a unique user-defined name along with the constraint definition. A constraint can then, be dropped by referring to the constraint by its name. Under these circumstances a user-defined constraint name becomes very convenient.

User named constraints simplifies the task of dropping constraints. A constraint can be given a user-defined name by preceding the constraint definition with the reserved word **CONSTRAINT** and a **user-defined name**.

Syntax:

```
CONSTRAINT <Constraint Name> <Constraint Definition>
```

Example 7:
Drop the CUST_MSTR table, if it already exists. Create a table CUST_MSTR with a primary key constraint on the column CUST_NO and also define its constraint name.

```
DROP TABLE CUST_MSTR;
CREATE TABLE CUST_MSTR (
    "CUST_NO" VARCHAR2(10) CONSTRAINT p_CUSTKey PRIMARY KEY,
    "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DOB_INC" DATE,
    "OCCUP" VARCHAR2(25), "PHOTOGRAPH" VARCHAR2(25),
    "SIGNATURE"  VARCHAR2(25), "PANCOPY" VARCHAR2(1),
    "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

Example 8:
Drop the table EMP_MSTR, if it already exists. Create a table EMP_MSTR with its foreign key as BRANCH_NO. The foreign key is BRANCH_NO available and as a primary key in the BRANCH_MSTR table. Also define the name of the foreign key.

```
DROP TABLE EMP_MSTR;
CREATE TABLE "DBA_BANKSYS"."EMP_MSTR"(
    "EMP_NO" VARCHAR2(10), "BRANCH_NO" VARCHAR2(10),
    "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DEPT" VARCHAR2(30),
    "DESIG" VARCHAR2(30),
    CONSTRAINT f_BranchKey
        FOREIGN KEY (BRANCH_NO) REFERENCES BRANCH_MSTR);
```

Output:

```
Table created.
```

The Unique Key Constraint

The **Unique** column constraint **permits multiple entries** of NULL into the column. These NULL values are clubbed at the top of the column in the order in which they were entered into the table. This is **the essential difference** between the Primary Key and the Unique constraints when applied to table column(s).

Key point about Unique Constraint:

1. Unique key will not allow duplicate values
2. Unique index is created automatically
3. A table can have more than one Unique key which is not possible in Primary key
4. Unique key can combine upto 16 columns in a Composite Unique key
5. Unique key can not be LONG or LONG RAW data type

UNIQUE Constraint Defined At The Column Level

Syntax:

```
<ColumnName> <Datatype>(<Size>) UNIQUE
```

Example 9:
Drop the CUST_MSTR table, if it already exists. Create a table CUST_MSTR such that the contents of the column CUST_NO are unique across the entire column.

```
DROP TABLE CUST_MSTR;
CREATE TABLE CUST_MSTR (
    "CUST_NO" VARCHAR2(10) UNIQUE, "FNAME" VARCHAR2(25), "MNAME" VARCHAR2(25),
    "LNAME" VARCHAR2(25), "DOB_INC" DATE, "OCCUP" VARCHAR2(25),
    "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25), "PANCOPY" VARCHAR2(1),
    "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

For testing the Unique constraint execute the following INSERT INTO statements:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C1', 'Ivan', 'Nelson', 'Bayross', '25-JUN-1952', 'Self Employed', 'D:/ClntPht/C1.gif',
           'D:/ClntSgnt/C1.gif', 'Y', 'Y');
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C1', 'Chriselle', 'Ivan', 'Bayross', '29-OCT-1982', 'Service', 'D:/ClntPht/C2.gif', 'D:/ClntSgnt/C2.gif', 'N',
           'Y');
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C2', 'Mamta', 'Arvind', 'Muzumdar', '28-AUG-1975', 'Service', 'D:/ClntPht/C3.gif', 'D:/ClntSgnt/C3.gif',
           'Y', 'Y');
```

Output:
The first INSERT INTO statement will execute without any errors as show below:

```
1 row created.
```

When the second INSERT INTO statement is executed an errors occurs as show below:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-00001: unique constraint (DBA_BANKSYS.SYS_C003007) violated
```

The third INSERT INTO statement rectifies this and the result is as show below:

```
1 row created.
```

When a SELECT statement is executed on the Client_Master table the records reterived are:

```
SELECT CUST_NO, FNAME, MNAME, LNAME FROM CUST_MSTR;
```

Output:

```
CUST_NO    FNAME    MNAME    LNAME
-------------------------------------
C1         Ivan     Nelson   Bayross
C2         Mamta    Arvind   Muzumdar
```


UNIQUE Constraint Defined At The Table Level

Syntax:

```
CREATE TABLE TableName
    (<ColumnName1> <Datatype>(<Size>), <ColumnName2> <Datatype>(<Size>),
     UNIQUE (<ColumnName1>, <ColumnName2>));
```

Example 10:
Drop the CUST_MSTR table, if it already exists. Create a table CUST_MSTR such that the contents of the column CUST_NO are unique across the entire column.

```
DROP TABLE CUST_MSTR;
CREATE TABLE CUST_MSTR (
    "CUST_NO" VARCHAR2(10), "FNAME" VARCHAR2(25),
    "MNAME" VARCHAR2(25), "LNAME" VARCHAR2(25),
    "DOB_INC" DATE, "OCCUP" VARCHAR2(25),
    "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25),
    "PANCOPY" VARCHAR2(1), "FORM60" VARCHAR2(1),
    UNIQUE(CUST_NO));
```

Output:

```
Table created.
```

In the case of the table level unique constraints, the result for the following INSERT INTO statement will remain the same as explained earlier.

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C1', 'Ivan', 'Nelson', 'Bayross', '25-JUN-1952', 'Self Employed', 'D:/ClntPht/C1.gif',
           D:/ClntSgnt/C1.gif', 'Y', 'Y');
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C1', 'Chriselle', 'Ivan', 'Bayross', '29-OCT-1982', 'Service', 'D:/ClntPht/C2.gif', 'D:/ClntSgnt/C2.gif', 'N',
           'Y');
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C2', 'Mamta', 'Arvind', 'Muzumdar', '28-AUG-1975', 'Service', 'D:/ClntPht/C3.gif', 'D:/ClntSgnt/C3.gif',
           'Y', 'Y');
```

Business Rule Constraints

Oracle allows the application of **business rules** to table columns. Business managers determine business rules, they vary from system to system as mentioned earlier. These rules are applied to data, **prior** the data is being inserted into table columns. This ensures that the data (**records**) in the table have integrity.

For example, the rule that no employee in the company shall get a salary less than Rs.1000/- is a business rule. This means that no cell in the **salary** column of the employee table should hold a **value** less than 1000. If an attempt is made, to insert a value less than 1000 into the salary column, the database engine rejects the entire record automatically.

Business rules can be implemented in Oracle by using **CHECK** constraints. Check Constraints can be bound to a **column** or a **table** using the **CREATE TABLE** or **ALTER TABLE** command.

Business rule validation checks are performed when any table **write** operation is carried out. Any insert or update statement causes the relevant Check constraint to be evaluated. The Check constraint must be satisfied for the write operation to succeed. Thus **Check constraints** ensure the integrity of the data in tables.

Conceptually, data constraints are connected to a column, by the Oracle engine, as **flags**. Whenever, an attempt is made to load the column with data, the Oracle engine observes the flag and recognizes the presence of a constraint. The Oracle engine then retrieves the Check constraint definition and then applies the Check constraint definition, to the data being loaded into the table column. If the data being entered into a column fails any of the data constraint checks, the **entire** record is rejected. The Oracle engine will then flash an appropriate **error message**.

Oracle allows programmers to define constraints at:

- Column Level
- Table Level

Column Level Constraints

If data constraints are defined as an attribute of a column definition when creating or altering a table structure, they are **column level constraints**.

Caution

Column level constraints are applied to the **current column**. The current column is the column that immediately **precedes** the constraint (**i.e.** they are local to a specific column). A column level constraint **cannot** be applied if the data constraint spans **across multiple columns** in a table.

Table Level Constraints

If data constraints are defined **after defining all table column attributes** when creating or altering a table structure, it is a **table level constraint**.

Note

A table level constraint **must** be applied if the data constraint **spans across multiple columns** in a table.

Constraints are stored as a part of the global table definition by the Oracle engine in its **system tables**. The SQL syntax used to attach the constraint will change depending upon whether it is a column level or table level constraint.

NULL Value Concepts

Often there may be records in a table that do not have values for every field. This could be because the information is not available at the time of data entry or because the field is not applicable in every case. If the column was created as **NULLABLE**, Oracle will place a NULL value in the column in the **absence** of a user-defined value.

A NULL value is **different from** a blank or a zero. A **NULL value** can be inserted into **columns** of **any data type**.

Principles Of NULL Values

- Setting a NULL value is appropriate when the actual value is unknown, or when a value would not be meaningful
- A NULL value is **not equivalent** to a value of **zero** if the data type is **number** and is not equivalent to **spaces** if the data type is **character**
- A NULL value will evaluate to NULL in any expression (**e.g.** NULL multiplied by 10 is NULL)
- NULL value can be inserted into columns of **any data type**
- If the column has a NULL value, Oracle ignores any UNIQUE, FOREIGN KEY, CHECK constraints that may be attached to the column

Difference Between An Empty String And A NULL Value

Oracle has changed its rules about empty strings and null values in newer versions of Oracle. Now, an empty string is treated as a null value in Oracle.

To understand this go through the following example:

Example 11:
Drop the BRANCH_MSTR table, if it already exists. Create a table BRANCH_MSTR such that the contents of the column CUST_NO are unique across the entire column.

```
DROP TABLE BRANCH_MSTR;
CREATE TABLE BRANCH_MSTR (
    "BRANCH_NO" VARCHAR2(10), "NAME" VARCHAR2(25));
```

Output:

```
Table created.
```

Next, insert two records into this table.

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B1', null);
```

Output:

```
1 row created.
```

```
INSERT INTO BRANCH_MSTR (BRANCH_NO, NAME) VALUES('B2', '');
```

Output:

```
1 row created.
```

The first statement inserts a record with a branch name that is null, while the second statement inserts a record with an empty string as a branch name.

Now, retrieve all rows with a branch name that is an empty string value as follows:

```
SELECT * FROM BRANCH_MSTR WHERE NAME = '';
```

When this statement is executed, it is expected to retrieve the row that was inserted above. But instead, this statement will not retrieve any records at all.

Now, try retrieving all rows where the branch name contains a null value:

```
SELECT * FROM BRANCH_MSTR WHERE NAME IS NULL;
```

When this statement is executed, both rows are retrieved. This is because Oracle has now changed its rules so that empty strings behave as null values.

It is also important to note that the null value is unique. Usual operands such as =, <, > and so on cannot be used on a null value. Instead, the IS NULL and IS NOT NULL conditions have to be used.

NOT NULL Constraint Defined At The Column Level

In addition to Primary key and Foreign Key, Oracle has **NOT NULL** as column constraint. The **NOT NULL** column constraint ensures that a table column cannot be left empty.

When a column is defined as **not null**, then that column becomes a **mandatory** column. It implies that a value must be entered into the column if the record is to be accepted for storage in the table.

Syntax:

<ColumnName> <Datatype>(<Size>) NOT NULL

Example 12:
Drop the table CUST_MSTR, if already exists and then create it again making Date of Birth field not null. Refer to the details of table in chapter 6

```
CREATE TABLE "DBA_BANKSYS"."CUST_MSTR"(
    "CUST_NO" VARCHAR2(10), "FNAME" VARCHAR2(25),
    "MNAME" VARCHAR2(25), "LNAME" VARCHAR2(25),
    "DOB_INC" DATE NOT NULL, "OCCUP" VARCHAR2(25),
    "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25),
    "PANCOPY" VARCHAR2(1), "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

Note

The **NOT NULL** constraint can only be applied at column level.

Execute the following INSERT INTO statements to verify whether mandatory field constraints are applied:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('O14', null, null, null, null, 'Retail Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-01400: cannot insert NULL into ("DBA_BANKSYS"."CUST_MSTR"."DOB_INC")
```

The above **error** message confirms that the mandatory field constraints are applied successfully.

Caution

The **NOT NULL** constraint can only be applied at column level. Although **NOT NULL** can be applied as a **CHECK** constraint, Oracle Corp recommends that this should **not be done**.

The CHECK Constraint

Business Rule validations can be applied to a table column by using **CHECK** constraint. **CHECK** constraints must be specified as a logical expression that evaluates either to **TRUE** or **FALSE**.

Note

A **CHECK** constraint takes substantially longer to execute as compared to NOT NULL, PRIMARY KEY, FOREIGN KEY or UNIQUE. Thus CHECK constraints must be avoided if the constraint can be defined using the Not Null, Primary key or Foreign key constraint.

CHECK constraint defined at the column level:

Syntax:

<ColumnName> <Datatype>(<Size>) CHECK (<Logical Expression>)

Example 13:
Drop the table CUST_MSTR, if already exists. Create a table **CUST_MSTR** with the following check constraints:

- Data values being inserted into the column **CUST_NO** must start with the capital letter **C**
- Data values being inserted into the column **FNAME, MNAME and LNAME** should be in **upper case** only

```
CREATE TABLE CUST_MSTR("CUST_NO" VARCHAR2(10) CHECK(CUST_NO LIKE 'C%'),
    "FNAME" VARCHAR2(25) CHECK (FNAME = UPPER(FNAME)),
    "MNAME" VARCHAR2(25) CHECK (MNAME = UPPER(MNAME)),
    "LNAME" VARCHAR2(25) CHECK (LNAME = UPPER(LNAME)), "DOB_INC" DATE,
    "OCCUP" VARCHAR2(25), "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25),
    "PANCOPY" VARCHAR2(1), "FORM60" VARCHAR2(1));
```

Output:

```
Table created.
```

CHECK Constraint Defined At The Table Level:

Syntax:

CHECK (<Logical Expression>)

Example 14:
Drop the table CUST_MSTR, if already exists. Create a table **CUST_MSTR** with the following check constraints:

- Data values being inserted into the column **CUST_NO** must start with the capital letter **C**
- Data values being inserted into the column **FNAME, MNAME and LNAME** should be in **upper case** only

```
CREATE TABLE CUST_MSTR("CUST_NO" VARCHAR2(10), "FNAME" VARCHAR2(25),
    "MNAME" VARCHAR2(25), "LNAME" VARCHAR2(25), "DOB_INC" DATE,
    "OCCUP" VARCHAR2(25), "PHOTOGRAPH" VARCHAR2(25), "SIGNATURE"  VARCHAR2(25),
    "PANCOPY" VARCHAR2(1), "FORM60" VARCHAR2(1),
    CHECK (CUST_NO LIKE 'C%'), CHECK (FNAME = UPPER(FNAME)),
    CHECK (MNAME = UPPER(MNAME)), CHECK (LNAME = UPPER(LNAME)));
```

Output:

```
Table created.
```

Execute the following INSERT INTO statements to verify whether check constraints are applied:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('O14', 'SHARANAM', 'CHAITANYA', 'SHAH', '03-Jan-1981', 'Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-02290: check constraint (DBA_BANKSYS.SYS_C003055) violated
```

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C14', 'sharanam', 'CHAITANYA', 'SHAH', '03-Jan-1981', 'Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-02290: check constraint (DBA_BANKSYS.SYS_C003056) violated
```

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C14', 'SHARANAM', 'Chaitanya', 'SHAH', '03-Jan-1981', 'Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-02290: check constraint (DBA_BANKSYS.SYS_C003057) violated
```

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP, PHOTOGRAPH,
                       SIGNATURE, PANCOPY, FORM60)
    VALUES('C14', 'SHARANAM', 'CHAITANYA', 'sHAh', '03-Jan-1981', 'Business', null, null, 'N', 'Y');
```

Output:

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, MNAME, LNAME, DOB_INC, OCCUP,
PHOTOGRAPH, SIGNATURE, PANCOPY,
*
ERROR at line 1:
ORA-02290: check constraint (DBA_BANKSYS.SYS_C003058) violated
```

The above **error** messages confirm that the check constraints are applied successfully.

When using **CHECK** constraints, consider the ANSI / ISO standard, which states that a CHECK constraint is violated only if the condition evaluates to **False**. A check constraint is not violated if the condition evaluates to **True**.

Note

If the expression in a check constraint does not return a **true / false**, the value is **Indeterminate** or **Unknown.** Unknown values do not violate a check constraint condition. For example, consider the following CHECK constraint for SellPrice column in the Product_Master table:

CHECK (SellPrice > 0)

At first glance, this rule may be interpreted as "do not allow a row in the **Product_Master** table unless **the Sellprice** is greater than 0". However, note that if a row is inserted with a **null SellPrice**, the row **does not violate** the CHECK constraint because the entire check condition is evaluated as **unknown**.

In this particular case, prevent such violations by placing the **not null** integrity constraint along with the check constraint on **SellPrice** column of the table **Product_Master**.

Restrictions On CHECK Constraints

A **CHECK** integrity constraint requires that a condition be **true** or **unknown** for the row to be processed. If an SQL statement causes the condition to evaluate to **false**, an appropriate error message is displayed and processing stops.

A **CHECK** constraint has the following limitations:

- The condition must be a **Boolean** expression that can be evaluated using the values in the row being inserted or updated.
- The condition cannot contain **subqueries** or **sequences**.
- The condition cannot include the SYSDATE, UID, USER or USERENV SQL functions.

DEFINING DIFFERENT CONSTRAINTS ON A TABLE

Example 15:

Drop the table FD_MSTR, if already exists. Create FD_MSTR table where

- The FD_SER_NO is a primary key to this table
- The BRANCH_NO is the foreign key referencing the table BRANCH_MSTR
- The CORP_CUST_NO is the foreign key referencing the table CUST_MSTR
- The VERI_EMP_NO is a foreign key referencing the table EMP_MSTR
- The CORP_CNST_TYPE will hold wither of the following values:
  ØS, 1C, 2C, 3C, 4C, 5C, 6C, 7C indicating different types of companies

```
CREATE TABLE "DBA_BANKSYS"."FD_MSTR"("FD_SER_NO" VARCHAR2(10),
    "SF_NO" VARCHAR2(10), "BRANCH_NO" VARCHAR2(10), "INTRO_CUST_NO" VARCHAR2(10),
    "INTRO_ACCT_NO" VARCHAR2(10), "INTRO_SIGN" VARCHAR2(1),
    "ACCT_NO" VARCHAR2(10), "TITLE" VARCHAR2(30), "CORP_CUST_NO" VARCHAR2(10),
    "CORP_CNST_TYPE" VARCHAR(4), "VERI_EMP_NO" VARCHAR2(10),
    "VERI_SIGN" VARCHAR2(1), "MANAGER_SIGN" VARCHAR2(1),
    CONSTRAINT PK PRIMARY KEY (FD_SER_NO, CORP_CUST_NO),
    CONSTRAINT FK_BR FOREIGN KEY (BRANCH_NO) REFERENCES BRANCH_MSTR,
    CONSTRAINT FK_CU FOREIGN KEY (CORP_CUST_NO) REFERENCES CUST_MSTR,
    CONSTRAINT FK_EM FOREIGN KEY (VERI_EMP_NO) REFERENCES EMP_MSTR,
    CONSTRAINT CHK CHECK (CORP_CNST_TYPE IN ('ØS', '1C', '2C', '3C', '4C', '5C', '6C', '7C')));
```

Output:

```
Table created.
```

THE USER_CONSTRAINTS TABLE

A table can be created with multiple constraints attached to its columns. If a user wishes to see the table structure along with its constraints, Oracle provides the **DESCRIBE <TableName>** command.

This command displays only the column names, data type, size and the NOT NULL constraint. The information about the other constraints that may be attached to the table columns such as the PRIMARY KEY, FOREIGN KEY, and so on, is not available using the DESCRIBE verb.

Oracle stores such information in a table called **USER_CONSTRAINTS**. Querying **USER_CONSTRAINTS** provides information bound to the names of all the constraints on the table. **USER_CONSTRAINTS** comprises of multiple columns, some of which are described below:

USER_CONSTRAINTS Table:

Column Name	Description
OWNER	The owner of the constraint.
CONSTRAINT_NAME	The name of the constraint
TABLE_NAME	The name of the table associated with the constraint
CONSTRAINT_TYPE	The type of constraint: **P**: Primary Key Constraint **R**: Foreign Key Constraint **U**: Unique Constraint **C**: Check Constraint
SEARCH_CONDITION	The search condition used (for CHECK Constraints)
R_OWNER	The owner of the table referenced by the FOREIGN KEY constraints
R_CONSTRAINT_NAME	The name of the constraint referenced by a FOREIGN KEY constraint.

Example 16:
View the constraints of the table CUST_MSTR

```
SELECT OWNER, CONSTRAINT_NAME, CONSTRAINT_TYPE FROM USER_CONSTRAINTS
    WHERE TABLE_NAME = 'CUST_MSTR';
```

Output:

```
OWNER          CONSTRAINT_NAME   CONSTRAINT_TYPE
DBA_BANKSYS    SYS_C003027       C
DBA_BANKSYS    SYS_C003028       C
DBA_BANKSYS    SYS_C003029       C
DBA_BANKSYS    SYS_C003030       C
```

DEFINING INTEGRITY CONSTRAINTS VIA THE ALTER TABLE COMMAND

Integrity constraints can be defined using the **constraint** clause, in the **ALTER TABLE** command.

Note

Oracle **will not allow** constraints defined using the **ALTER TABLE**, to be applied to the table if data previously placed in the table **violates such constraints**.

If a Primary key constraint was being applied to a table in retrospect and the column has duplicate values in it, the Primary key constraint **will not** be set to that column.

The following examples show the definitions of several integrity constraints:

Example 17:
Alter the table EMP_MSTR by adding a primary key on the column EMP_NO.

```
ALTER TABLE EMP_MSTR ADD PRIMARY KEY (EMP_NO);
```

Output:

```
Table altered.
```

Example 18:
Add FOREIGN KEY constraint on the column **VERI_EMP_NO** belonging to the table **FD_MSTR**, which references the table **EMP_MSTR**. Modify column **MANAGER_SIGN** to include the **NOT NULL** constraint

```
ALTER TABLE FD_MSTR ADD CONSTRAINT F_EmpKey FOREIGN KEY(VERI_EMP_NO)
    REFERENCES EMP_MSTR MODIFY(MANAGER_SIGN NOT NULL);
```

Output:

```
Table altered.
```

DROPPING INTEGRITY CONSTRAINTS VIA THE ALTER TABLE COMMAND

Integrity constraint can be dropped if the rule that it enforces is no longer **true** or if the constraint is no longer **needed**. Drop the constraint using the **ALTER TABLE** command with the **DROP** clause. The following examples illustrate the dropping of integrity constraints:

Example 19:
Drop the PRIMARY KEY constraint from EMP_MSTR.

```
ALTER TABLE EMP_MSTR DROP PRIMARY KEY;
```

Output:

```
Table altered.
```

Example 20:
Drop FOREIGN KEY constraint on column VERI_EMP_NO in table FD_MSTR

```
ALTER TABLE FD_MSTR DROP CONSTRAINT F_EmpKey;
```

Output:

```
Table altered.
```

Note

Dropping UNIQUE and PRIMARY KEY constraints **also drops all** associated indexes.

DEFAULT VALUE CONCEPTS

At the time of table creation a **default value** can be assigned to a column. When a record is loaded into the table, and the column is left empty, the Oracle engine will automatically load this column with the default value specified. The data type of the default value should match the data type of the column. The **DEFAULT** clause can be used to specify a default value for a column.

Syntax:

```
<ColumnName> <Datatype>(<Size>) DEFAULT <Value>;
```

Example 21:
Create ACCT_MSTR table where the column CURBAL is the number and by default it should be zero. The other column STATUS is a varchar2 and by default it should have character **A**. (Refer to table definitions in the chapter 6)

```
CREATE TABLE "DBA_BANKSYS"."ACCT_MSTR"("ACCT_NO" VARCHAR2(10),
    "SF_NO" VARCHAR2(10), "LF_NO" VARCHAR2(10), "BRANCH_NO" VARCHAR2(10),
    "INTRO_CUST_NO" VARCHAR2(10), "INTRO_ACCT_NO" VARCHAR2(10),
    "INTRO_SIGN" VARCHAR2(1), "TYPE" VARCHAR2(2), "OPR_MODE" VARCHAR2(2),
    "CUR_ACCT_TYPE" VARCHAR2(4), "TITLE" VARCHAR2(30),
    "CORP_CUST_NO" VARCHAR2(10), "APLNDT" DATE, "OPNDT" DATE,
    "VERI_EMP_NO" VARCHAR2(10), "VERI_SIGN" VARCHAR2(1),
    "MANAGER_SIGN" VARCHAR2(1), "CURBAL" NUMBER(8, 2) DEFAULT 0,
    "STATUS" VARCHAR2(1) DEFAULT 'A');
```

Output:

```
Table created.
```

Note

- The data type of the default value should match the data type of the column
- Character and date values will be specified in single quotes
- If a column level constraint is defined on the column with a default value, the default value clause must precede the constraint definition

Thus the syntax will be:

```
<ColumnName> <Datatype>(<Size>) DEFAULT <Value> <constraint definition>
```

SELF REVIEW QUESTIONS

FILL IN THE BLANKS

1. Business rules, which are enforced on data being stored in a table, are called _____.
2. If the column was created as _____ Oracle will place a NULL value in the column in the absence of a user-defined value.
3. When a column is defined as not null, then that column becomes a _____ column.
4. The _____ constraint can only be applied at column level.
5. A _____ value can be inserted into the columns of any data type.
6. A single column primary key is called a _____ key.
7. The data held across the primary key column must be _____.
8. _____ keys represent relationships between tables.
9. The table in which the foreign key is defined is called a Foreign table or _____ table.

10. The default behavior of the foreign key can be changed by using the _____ option.
11. _____ constraints must be specified as a logical expression that evaluates either to TRUE or FALSE.
12. In a CHECK constraint the condition must be a _____ expression that can be evaluated using the values in the row being inserted or updated.
13. _____ constraints can be defined using the constraint clause, in the ALTER TABLE command.
14. Dropping UNIQUE and PRIMARY KEY constraints also drops all associated _____.

TRUE OR FALSE

15. Business rules that have to be applied to data are completely System dependent.
16. Constraints super control the data being entered into a table for temporary storage.
17. A NULL value is equivalent to a value of zero.
18. Setting a NULL value is appropriate when the actual value is unknown.
19. A table cannot contain multiple unique keys.
20. Oracle ignores any UNIQUE, FOREIGN KEY, CHECK constraints on a NULL value.
21. A primary key column in a table is an optional column.
22. Standard business rules do not allow multiple entries for the same product.
23. The master table can be referenced in the foreign key definition by using the clause REFERENCES tablename.columname when defining the foreign key.
24. A CHECK constraint consists of subqueries and sequences.
25. The USER_CONSTRAINTS command displays only the column names, data type, size and the NOT NULL constraint.
26. Drop the constraint using the DROP TABLE command with the DELETE clause.
27. At the time of table creation a default value can be assigned to a column.
28. If a column level constraint is defined on the column with a default value, the default value clause must precede the constraint definition.

HANDS ON EXERCISES

1. Create the tables described below:

Table Name: CLIENT_MASTER
Description: Used to store client information.

Column Name	Data Type	Size	Default	Attributes
CLIENTNO	Varchar2	6		Primary Key / first letter must start with 'C'
NAME	Varchar2	20		Not Null
ADDRESS1	Varchar2	30		

Details for **CLIENT_MASTER** table continued.

Column Name	Data Type	Size	Default	Attributes
ADDRESS2	Varchar2	30		
CITY	Varchar2	15		
PINCODE	Number	8		
STATE	Varchar2	15		
BALDUE	Number	10,2		

Table Name: **PRODUCT_MASTER**
Description: Used to store product information.

Column Name	Data Type	Size	Default	Attributes
PRODUCTNO	Varchar2	6		Primary Key / first letter must start with 'P'
DESCRIPTION	Varchar2	15		Not Null
PROFITPERCENT	Number	4,2		Not Null
UNITMEASURE	Varchar2	10		Not Null
QTYONHAND	Number	8		Not Null
REORDERLVL	Number	8		Not Null
SELLPRICE	Number	8,2		Not Null, Cannot be 0
COSTPRICE	Number	8,2		Not Null, Cannot be 0

Table Name: **SALESMAN_MASTER**
Description: Used to store salesman information working for the company.

Column Name	Data Type	Size	Default	Attributes
SALESMANNO	Varchar2	6		Primary Key / first letter must start with 'S'
SALESMANNAME	Varchar2	20		Not Null
ADDRESS1	Varchar2	30		Not Null
ADDRESS2	Varchar2	30		
CITY	Varchar2	20		
PINCODE	Number	8		
STATE	Varchar2	20		
SALAMT	Number	8,2		Not Null, Cannot be 0
TGTTOGET	Number	6,2		Not Null, Cannot be 0
YTDSALES	Number	6,2		Not Null
REMARKS	Varchar2	60		

Table Name: **SALES_ORDER**
Description: Used to store client's orders.

Column Name	Data Type	Size	Default	Attributes
ORDERNO	Varchar2	6		Primary Key / first letter must start with 'O'
CLIENTNO	Varchar2	6		Foreign Key references ClientNo of Client_Master table
ORDERDATE	Date			Not Null
DELYADDR	Varchar2	25		
SALESMANNO	Varchar2	6		Foreign Key references SalesmanNo of Salesman_Master table
DELYTYPE	Char	1	F	Delivery: part (P) / full (F)
BILLYN	Char	1		
DELYDATE	Date			Cannot be less than Order_Date
ORDERSTATUS	Varchar2	10		Values ('In Process', 'Fulfilled', 'BackOrder', 'Cancelled')

Check orderstatus In( ).

Table Name: **SALES_ORDER_DETAILS**
Description: Used to store client's orders with details of each product ordered.

Column Name	Data Type	Size	Default	Attributes
ORDERNO	Varchar2	6		Foreign Key references OrderNo of Sales_Order table
PRODUCTNO	Varchar2	6		Foreign Key references ProductNo of Product_Master table
QTYORDERED	Number	8		
QTYDISP	Number	8		
PRODUCTRATE	Number	10,2		

2. Insert the following data into their respective tables:

a) Re-insert the data generated for tables CLIENT_MASTER, PRODUCT_MASTER, and SALESMAN_MASTER. Refer to hands-on exercised for **Chapter 07:Interactive SQL-Part I**.

b) Data for Sales_Order table:

OrderNo	ClientNo	OrderDate	SalesmanNo	DelyType	BillYN	DelyDate	OrderStatus
O19001	C00001	12-June-04	S00001	F	N	20-July-02	In Process
O19002	C00002	25-June-04	S00002	P	N	27-June-02	Cancelled
O46865	C00003	18-Feb-04	S00003	F	Y	20-Feb-02	Fulfilled
O19003	C00001	03-Apr-04	S00001	F	Y	07-Apr-02	Fulfilled
O46866	C00004	20-May-04	S00002	P	N	22-May-02	Cancelled
O19008	C00005	24-May-04	S00004	F	N	26-July-02	In Process

c) Data for Sales_Order_Details table:

OrderNo	ProductNo	QtyOrdered	QtyDisp	ProductRate
O19001	P00001	4	4	525
O19001	P07965	2	1	8400
O19001	P07885	2	1	5250
O19002	P00001	10	0	525
O46865	P07868	3	3	3150
O46865	P07885	3	1	5250
O46865	P00001	10	10	525
O46865	P0345	4	4	1050
O19003	P03453	2	2	1050
O19003	P06734	1	1	12000
O46866	P07965	1	0	8400
O46866	P07975	1	0	1050
O19008	P00001	10	5	525
O19008	P07975	5	3	1050

9. INTERACTIVE SQL PART - III

COMPUTATIONS DONE ON TABLE DATA

None of the techniques used till now allows display of data from a table after some arithmetic has been done with it.

Computations may include displaying an employee's name and the employee's salary from the Employee_Master table along with the **annual salary** of the employee (**i.e.** Salary*12). The arithmetic (Salary * 12) is an example of table data arithmetic.

Arithmetic and logical operators give a new dimension to SQL sentences.

Arithmetic Operators

Oracle allows arithmetic operators to be used while viewing records from a table or while performing Data Manipulation operations such as Insert, Update and Delete. These are:

+	Addition	*	Multiplication
-	Subtraction	**	Exponentiation
/	Division	()	Enclosed operation

Example 1:
List the fixed deposits held by the customers and also show what will be the amount payable by the bank if the fixed deposits are cancelled by the **end of the day**.

Synopsis:

Tables:	FD_DTLS
Columns:	FD_NO, TYPE, PERIOD, OPNDT, DUEDT, AMT, INTRATE, DUEAMT
Technique:	**Functions:** ROUND(), **Operators:** *, -, /, **Clauses:** WHERE, **Others:** SYSDATE

Solution:

```
SELECT FD_NO, TYPE, PERIOD, OPNDT, DUEDT, AMT, INTRATE, DUEAMT,
       ROUND(AMT + (AMT * ROUND(SYSDATE – OPNDT)/365 * (INTRATE/100)), 2)
   FROM FD_DTLS WHERE DUEDT > SYSDATE;
```

Output:

```
FD_NO   TYPE    PERIOD OPNDT      DUEDT          AMT  INTRATE   DUEAMT
ROUND(AMT+(AMT*ROUND(SYSDATE-OPNDT)/365*(INTRATE/100)),2)
F6      S          732 19-JUL-03 20-JUL-05     5000         9  5902.47
                                                      5429.04
F7      S          366 27-JUL-03 27-JUL-04     5000         8  5401.1
                                                      5372.6
```

Explanation:
Here, **ROUND(AMT + (AMT * ROUND(SYSDATE – OPNDT)/365 * (INTRATE/100)),2)** is **not** a column in the table **FD_DTLS**. However, the arithmetic specified is done on the contents of the columns **AMT, OPNDT and INTRATE** of the table **FD_DTLS** and displayed in the output of the query.

By default, the Oracle engine will use the column names of the table **FD_DTLS** as column headers when displaying column output on the VDU screen.

Since there are no columns with the arithmetic expression applied on the table **FD_DTLS**, the Oracle engine performs the required arithmetic and uses the **formula** as the **default** column header when displaying output as seen above.

Renaming Columns Used With Expression Lists

Rename the default output column names with an **alias**, when required.

Syntax:

```
SELECT <ColumnName> <AliasName>, <ColumnName> <AliasName>
   FROM <TableName>;
```

Example 2:
List the fixed deposits held by the customers and also show what will be the amount received if the fixed deposits are cancelled on the same day. Use Alias to rename the calculative column to **Pre-Maturity Amount**.

Synopsis:

Tables:	FD_DTLS
Columns:	FD_NO, TYPE, PERIOD, OPNDT, DUEDT, AMT, INTRATE, DUEAMT
Technique:	**Functions:** ROUND(), **Operators:** *, -, /, **Clauses:** WHERE, **Others:** ALIAS, SYSDATE

Solution:

```
SELECT FD_NO, TYPE, PERIOD, OPNDT, DUEDT, AMT, INTRATE, DUEAMT,
       ROUND(AMT + (AMT * ROUND(SYSDATE - OPNDT)/365 * (INTRATE/100)), 2)
          "Pre-Maturity Amount"
   FROM FD_DTLS WHERE DUEDT > SysDate;
```

Output:

```
FD_NO   TYPE    PERIOD OPNDT      DUEDT          AMT  INTRATE   DUEAMT
 Pre Maturity Amount
F6      S          732 19-JUL-03 20-JUL-05     5000         9  5902.47
                                                      5429.04
F7      S          366 27-JUL-03 27-JUL-04     5000         8  5401.1
                                                      5372.6
```

Explanation:
Here, **ROUND(AMT + (AMT * ROUND(SYSDATE – OPNDT)/365 * (INTRATE/100)),2)** is renamed to alias "Pre-Maturity Amount".

Logical Operators

Logical operators that can be used in SQL sentences are:

The AND Operator:

The AND operator allows creating an SQL statement based on two or more conditions being met. It can be used in any valid SQL statement such as select, insert, update, or delete. The AND operator requires that each condition must be met for the record to be included in the result set.

The Oracle engine will process all rows in a table and display the result only when **all** of the conditions specified using the **AND** operator are satisfied.

Example 3:
Display all those transactions performed today for amount ranging between 500 and 5000 **both inclusive**.

Synopsis:

Tables:	TRANS_MSTR
Columns:	All Columns
Technique:	**Functions:** TO_CHAR(), **Operators:** AND, **Clauses:** WHERE, **Others:** SYSDATE

Solution:

```
SELECT * FROM TRANS_MSTR WHERE AMT >= 500 AND AMT <= 5000
    AND TO_CHAR(DT, 'DD/MM/YYYY') = TO_CHAR(SYSDATE, 'DD/MM/YYYY');
```

Output:

```
TRANS_NO ACCT_NO DT          TYPE PARTICULAR       DR_CR   AMT  BALANCE
-------- ------- ----------- ---- ---------------- ------ ----- --------
T7       CA7     14-MAR-2004 B    Initial Payment  D      2000     2000
```

Explanation:
Here, the AND operator is used to compare the value held in the amount field with a constant. Only those transactions carried out today, that satisfy this comparison, are shown. This is done by comparing transaction dates with the current date after converting them to characters.

The OR Operator:

The OR condition allows creating an SQL statement where records are returned when any one of the conditions are met. It can be used in any valid SQL statement such as select, insert, update, or delete. The OR condition requires that any of the conditions must be met for the record to be included in the result set.

The Oracle engine will process all rows in a table and display the result only when **any of** the conditions specified using the **OR** operator is satisfied.

Example 4:
Display the customers whose belong to Information Technology or are self-employed.

Synopsis:

Tables:	CUST_MSTR, ADDR_DTLS
Columns:	CUST_NO, FNAME, MNAME, LNAME
Technique:	**Operators:** LIKE, AND, OR, **Clauses:** WHERE, **Others:** CONCAT

Solution:

```
SELECT CUST_NO, FNAME || ' ' || MNAME || ' ' || LNAME "Customers"
    FROM CUST_MSTR, ADDR_DTLS
        WHERE CUST_MSTR.CUST_NO = ADDR_DTLS.CODE_NO
            AND (OCCUP = 'Information Technology' OR OCCUP = 'Self Employed')
            AND CUST_NO LIKE 'C%';
```

Output:

```
CUST_NO  Customers
-------- ------------------
C1       Ivan Nelson Bayross
C10      Namita S. Kanade
C7       Anil Arun Dhone
```


Explanation:
Here, the **OR** operator is used to compare the value held in the **OCCUP** field. If the comparison condition is satisfied then only those customers who belong to Information Technology or are self-employed are shown. The **LIKE** operator is used to avoid display of those rows held in the **CUST_MSTR** table, which identify corporates.

Combining the AND and OR Operator:

The **AND** and **OR** conditions can be combined in a single SQL statement. It can be used in any valid SQL statement such as select, insert, update, or delete.

When combining these conditions, it is important to use brackets so that the database knows what order to evaluate each condition.

The Oracle engine will process all rows in a table and display the result only when **all** of the conditions specified using the **AND** operator are satisfied and when **any of** the conditions specified using the **OR** operator are satisfied.

Example 5:
Display all the customers whose **last name** is **Bayross** and are **less than 25** yrs old **or** all those customers who are **more than 25** but **less than 50** yrs old.

Synopsis:

Tables:	CUST_MSTR, ADDR_DTLS
Columns:	CUST_NO, FNAME, MNAME, LNAME
Technique:	**Operators:** LIKE, AND, OR, **Clauses:** WHERE, **Others:** CONCATENATE

Solution:

```
SELECT CUST_NO, FNAME || ' ' || MNAME || ' ' || LNAME "Customers",
        ROUND((SYSDATE - DOB_INC)/365) "Age" FROM CUST_MSTR
    WHERE (ROUND((SYSDATE - DOB_INC)/365) < 25 AND LNAME='Bayross')
        OR (ROUND((SYSDATE - DOB_INC)/365) > 25
            AND ROUND((SYSDATE - DOB_INC)/365) < 50) AND CUST_NO LIKE 'C%';
```

Output:

```
CUST_NO  Customers                Age
-------- ------------------------ ---
C2       Chriselle Ivan Bayross   22
C3       Mamta Arvind Muzumdar    29
C4       Chhaya Sudhakar Bankar   28
C5       Ashwini Dilip Joshi      26
C8       Alex Austin Fernandes    42
C10      Namita S. Kanade         26
6 rows selected.
```

Explanation:
This would return all the records where the value calculated by the arithmetic expression **i.e.** age is less than 25 and the value held in the field **LNAME** is **Bayross**. This will also return those records where the value calculated by the arithmetic expression **i.e.** age is more than 25 but less than 50. The brackets determine what order the **AND / OR** conditions are evaluated in.

The NOT Operator:

The Oracle engine will process all rows in a table and display only those records that **do not** satisfy the condition specified.

Example 6:
List the accounts details of those accounts which are **neither** Singly and **nor** Joint Accounts.

Synopsis:

Tables:	ACCT_MSTR
Columns:	ACCT_NO, TYPE, OPR_MODE, OPNDT, CURBAL, STATUS
Technique:	**Operators:** NOT, OR, **Clauses:** WHERE

Solution:
```
SELECT ACCT_NO, TYPE, OPR_MODE, OPNDT, CURBAL, STATUS
   FROM ACCT_MSTR WHERE NOT (OPR_MODE = 'SI' OR OPR_MODE = 'JO');
```

Output:

ACCT_NO	TYPE	OPR_MODE	OPNDT	CURBAL	STATUS
CA4	CA	AS	05-FEB-03	2000	A
SB6	SB	ES	27-FEB-03	500	A
CA7	CA	AS	14-MAR-03	2000	A
CA10	CA	AS	19-APR-03	2000	A

Explanation:
The Oracle engine **will not** display rows from the ACCT_MSTR table where the value of the field **OPR_MODE** is either **SI** (Single) or **JO** (Joint). This means that all those records, which satisfy the condition specified using the NOT operator, will not be shown.

Range Searching

In order to select data that is within a range of values, the **BETWEEN** operator is used. The **BETWEEN** operator allows the selection of rows that contain values within a specified lower and upper limit. The range coded after the word **BETWEEN** is **inclusive**.

The lower value must be coded first. The two values in between the range must be linked with the keyword **AND**. The **BETWEEN** operator can be used with both character and numeric data types. However, the data types cannot be mixed **i.e.** the lower value of a range of values from a character column and the other from a numeric column.

Example 7:
List the transactions performed in months of January to March.

Synopsis:

Tables:	TRANS_MSTR
Columns:	All Columns
Technique:	**Functions:** TO_CHAR(),**Operators:** BETWEEN, **Clauses:** WHERE

Solution:
```
SELECT * FROM TRANS_MSTR WHERE TO_CHAR(DT, 'MM') BETWEEN 01 AND 03;
```

Equivalent to:
```
SELECT * FROM TRANS_MSTR
   WHERE TO_CHAR(DT, 'MM') >= 01 AND TO_CHAR(DT, 'MM') <= 03;
```


Output:

TRANS_NO	ACCT_NO	DT	TYPE	PARTICULAR	DR_CR	AMT	BALANCE
T1	SB1	05-JAN-03	C	Initial Payment	D	500	500
T2	CA2	10-JAN-03	C	Initial Payment	D	2000	2000
T3	SB3	22-JAN-03	C	Initial Payment	D	500	500
T4	CA4	05-FEB-03	B	Initial Payment	D	2000	2000
T5	SB5	15-FEB-03	B	Initial Payment	D	500	500
T6	SB6	27-FEB-03	C	Initial Payment	D	500	500
T7	CA7	14-MAR-03	B	Initial Payment	D	2000	2000
T8	SB8	29-MAR-03	C	Initial Payment	D	500	500

8 rows selected.

Explanation:
The above select will retrieve all those records from the ACCT_MSTR table where the value held in the DT field is between 01 and 03 (both values inclusive). This is done using TO_CHAR() function which extracts the month value from the DT field. This is then compared using the AND operator.

Example 8:
List all the accounts, which have not been accessed in the fourth quarter of the financial year.

Synopsis:

Tables:	TRANS_MSTR
Columns:	ACCT_NO
Technique:	**Functions:** TO_CHAR(),**Operators:** NOT, BETWEEN, **Clauses:** WHERE

Solution:
```
SELECT DISTINCT
 FROM TRANS_MSTR
    WHERE TO_CHAR(DT, 'MM') NOT BETWEEN 01 AND 04;
```

Output:

ACCT_NO
SB9

Explanation:
The above select will retrieve all those records from the ACCT_MSTR table where the value held in the DT field is not between 01 and 04 (both values inclusive). This is done using TO_CHAR() function which extracts the month value from the DT field and then compares them using the not and the between operator.

Pattern Matching

The use of the LIKE predicate

The comparison operators discussed so far have compared one value, exactly to one other value. Such precision may not always be desired or necessary. For this purpose Oracle provides the **LIKE** predicate.

The **LIKE** predicate allows comparison of one string value with another string value, which is not identical. This is achieved by using wildcard characters. Two wildcard characters that are available are:

For character data types:
- **%** allows to match any string of any length (including zero length)
- **_** allows to match on a single character

Example 9:
List the customers whose names begin with the letters 'Ch'.

Synopsis:	
Tables:	CUST_MSTR
Columns:	FNAME, LNAME, DOB_INC
Technique:	**Operators:** LIKE, **Clauses:** WHERE, **Others:** ALIAS

Solution:
SELECT FNAME, LNAME, DOB_INC "BIRTHDATE", OCCUP **FROM** CUST_MSTR
WHERE FNAME **LIKE** 'Ch%';

Output:

FNAME	LNAME	Birthday	OCCUP
Chriselle	Bayross	29-OCT-82	Service
Chhaya	Bankar	06-OCT-76	Service

Explanation:
In the above example, all those records where the value held in the field FNAME begins with **Ch** are displayed. The **%** indicates that any number of characters can follow the letters **Ch**.

Example 10:
List the customers whose names have the second character as **a** or **s**.

Synopsis:	
Tables:	CUST_MSTR
Columns:	FNAME, LNAME, DOB_INC
Technique:	**Operators:** LIKE, **Clauses:** WHERE, **Others:** ALIAS

Solution:
SELECT FNAME, LNAME, DOB_INC "Birthday", OCCUP **FROM** CUST_MSTR
WHERE FNAME **LIKE** '_a%' **OR** FNAME **LIKE** '_s%';

Output:

FNAME	LNAME	Birthday	OCCUP
Mamta	Muzumdar	28-AUG-75	Service
Ashwini	Joshi	20-NOV-78	Business
Hansel	Colaco	01-JAN-82	Service
Ashwini	Apte	19-APR-79	Service
Namita	Kanade	10-JUN-78	Self Employed

Explanation:
In the above example, all those records where the value held in the field FNAME contains the second character as **a** or **s** are displayed. The **_a** and **_s** indicates that only one character can precede the character a or s. The **%** indicates that any number of characters can follow the letters **Ch**.

Example 11:
List the customers whose names begin with the letters **Iv** and it is a four letter word.

Synopsis:	
Tables:	CUST_MSTR
Columns:	FNAME, LNAME, DOB_INC
Technique:	**Operators:** LIKE, **Clauses:** WHERE, **Others:** ALIAS

Solution:
SELECT FNAME, LNAME, DOB_INC "Birthday", OCCUP **FROM** CUST_MSTR
WHERE FNAME **LIKE** 'Iv__'; **(i.e. two underscore characters)**

Output:

FNAME	LNAME	Birthday	OCCUP
Ivan	Bayross	25-JUN-52	Self Employed

Explanation:
In the above example, all those records where the value held in the field FNAME begins with **Iv** are displayed. The __ **(i.e. two underscore characters)** indicates that only two characters can follow the letters **Iv**. This means the whole word will only be four characters.

The IN and NOT IN predicates:

The arithmetic operator **(=)** compares a single value to another single value. In case a value needs to be compared to a list of values then the **IN** predicate is used. The IN predicate helps reduce the need to use multiple OR conditions

Example 12:
List the customer details of the customers named Hansel, Mamta, Namita and Aruna.

Synopsis:

Tables:	CUST_MSTR
Columns:	FNAME, LNAME, DOB_INC
Technique:	**Operators:** IN, **Clauses:** WHERE, **Others:** ALIAS

Solution:
SELECT FNAME, LNAME, DOB_INC "birthday", OCCUP **FROM** CUST_MSTR
WHERE FNAME **IN**('Hansel', 'Mamta', 'Namita', 'Aruna');

Output:

FNAME	LNAME	Birthday	OCCUP
Mamta	Muzumdar	28-AUG-75	Service
Hansel	Colaco	01-JAN-82	Service
Namita	Kanade	10-JUN-78	Self Employed

Explanation:
The above example, displays all those records where the FNAME field holds any one of the four specified values.

The NOT IN predicate is the opposite of the IN predicate. This will select all the rows where values do not match the values in the list.

Example 13:
List the customer details of the customers other then Hansel, Mamta, Namita and Aruna.

Synopsis:

Tables:	CUST_MSTR
Columns:	FNAME, LNAME, DOB_INC
Technique:	**Operators:** NOT, IN, **Clauses:** WHERE, **Others:** ALIAS

Solution:

```
SELECT FNAME, LNAME, DOB_INC "Birthday", OCCUP FROM CUST_MSTR
   WHERE FNAME NOT IN('Hansel', 'Mamta', 'Namita', 'Aruna');
```

Output:

```
FNAME      LNAME      Birthday  OCCUP
---------- ---------- --------- -------------
Ivan       Bayross    25-JUN-52 Self Employed
Chriselle  Bayross    29-OCT-82 Service
Chhaya     Bankar     06-OCT-76 Service
Ashwini    Joshi      20-NOV-78 Business
Anil       Dhone      12-OCT-83 Self Employed
Alex       Fernandes  30-SEP-62 Executive
Ashwini    Apte       19-APR-79 Service
7 rows selected.
```

Explanation:

In the above example by just changing the predicate to **NOT IN** the Select statement will now retrieve all the rows where the FNAME is **not in** the values specified. In other words, information about customers whose names are **not Hansel, Mamta, Namita, Aruna** will be displayed.

The Oracle Table - DUAL

DUAL is a table owned by SYS. SYS owns the data dictionary, and DUAL is part of the data dictionary. Dual is a small Oracle worktable, which consists of only one row and one column, and contains the value **x** in that column. Besides arithmetic calculations, it also supports **date** retrieval and it's formatting.

Often a simple calculation needs to be done, for example, 2*2. The only SQL verb to cause an output to be written to a VDU screen is **SELECT**. However, a SELECT must have a table name in its FROM clause, otherwise the SELECT **fails**.

When an arithmetic exercise is to be performed such as 2*2 or 4/2 and so on, there is no table being referenced, only **numeric literals** are being used.

To facilitate such calculations via a SELECT, Oracle provides a **dummy** table called **DUAL**, against which SELECT statements that are required to manipulate numeric literals can be fired, and appropriate output obtained.

The structure of the dual table if viewed is as follows:

```
DESC DUAL;
```

Output:

```
NAME        Null?    TYPE
----------- -------- -------------
Dummy                VARCHAR2(1)
```

If the dual table is queried for records the output is as follows:

```
SELECT * FROM DUAL;
```

Output:

```
D
-
X
```

Example 14:

```
SELECT 2*2 FROM DUAL;
```


Output:

```
       2*2
----------
         4
```

SYSDATE

SYSDATE is a pseudo column that contains the current date and time. It requires no arguments when selected from the table DUAL and returns the current date.

Example 15:

```
SELECT SYSDATE FROM DUAL;
```

Output:

```
SYSDATE
---------
01-JUL-04
```

ORACLE FUNCTIONS

Oracle Functions serve the purpose of manipulating data items and returning a result. Functions are also capable of accepting user-supplied variables or constants and operating on them. Such variables or constants are called **arguments**. Any number of arguments (**or** no arguments at all) can be passed to a function in the following format:

Function_Name(argument1, argument2,..)

Oracle Functions can be clubbed together depending upon whether they operate on a single row or a group of rows retrieved from a table. Accordingly, functions can be classified as follows:

Group Functions (Aggregate Functions)

Functions that act on a **set of values** are called **Group Functions**. For example, **SUM**, is a function, which calculates the total set of numbers. A group function returns a single result row for a group of queried rows.

Scalar Functions (Single Row Functions)

Functions that act on **only one value** at a time are called **Scalar Functions**. For example, **LENGTH**, is a function, which calculates the length of one particular string value. A single row function returns one result for every row of a queried table or view.

Single row functions can be further grouped together by the data type of their arguments and return values. For example, **LENGTH** relates to the **String** Data type. Functions can be classified corresponding to different data types as:

String Functions : For **String** Data type
Numeric Functions: For **Number** Data type
Conversion Functions: For **Conversion** of one Data type to another.
Date Functions: For **Date** Data type

Aggregate Functions

AVG: Returns an average value of **'n'**, **ignoring** null values in a column.

Syntax:

```
AVG ([<DISTINCT>|<ALL>] <n>)
```

Example:
SELECT AVG(CURBAL) "Average Balance" **FROM** ACCT_MSTR;

Output:

Average Balance
1100

Note

In the above SELECT statement, **AVG** function is used to calculate the average balance of all accounts branch wise. The selected column is renamed as **Average Balance** in the output.

MIN: Returns a minimum value of **expr**.

Syntax:

```
MIN([<DISTINCT>|<ALL>] <expr>)
```

Example:
SELECT MIN(CURBAL) "Minimum Balance" **FROM** ACCT_MSTR;

Output:

Minimum Balance
500

COUNT(expr): Returns the number of rows where **expr** is not null.

Syntax:

```
COUNT([<DISTINCT>|<ALL>] <expr>)
```

Example:
SELECT COUNT(ACCT_NO) "No. Of Accounts" **FROM** ACCT_MSTR;

Output:

No. Of Accounts
10

COUNT(*): Returns the number of rows in the table, including duplicates and those with nulls.

Syntax:

```
COUNT(*)
```

Example:
SELECT COUNT(*) "No. Of Records" **FROM** ACCT_MSTR;

Output:

No. of Records
10

MAX: Returns the maximum value of **expr**.

Syntax:

```
MAX([<DISTINCT>|<ALL>] <expr>)
```

Example:
SELECT MAX(CURBAL) "Maximum Balance" **FROM** ACCT_MSTR;

Output:

Maximum Balance
2000

SUM: Returns the sum of the values of **'n'**.

Syntax:

```
SUM([<DISTINCT>|<ALL>] <n>)
```

Example:
SELECT SUM(CURBAL) "Total Balance" **FROM** ACCT_MSTR;

Output:

Total Balance
11000

Numeric Functions

ABS: Returns the absolute value of **'n'**.

Syntax:

```
ABS(n)
```

Example:
SELECT ABS(-15) "Absolute" **FROM DUAL;**

Output:

Absolute
15

POWER: Returns **m** raised to the **n^{th}** power. **n** must be an integer, else an error is returned.

Syntax:

```
POWER(m,n)
```

Example:
SELECT POWER(3,2) "Raised" **FROM DUAL;**

Output:

Raised
9

ROUND: Returns **n**, rounded to **m** places to the right of a decimal point. If **m** is omitted, **n** is rounded to **0** places. **m** can be negative to round off digits to the left of the decimal point. **m** must be an integer.

Syntax:

```
ROUND(n[,m])
```

Example:
SELECT ROUND(15.19,1) "Round" **FROM DUAL;**

Output:

Round
15.2

SQRT: Returns square root of **n**. If **n**<0, NULL. SQRT returns a **real** result.

Syntax:

```
SQRT(n)
```

Example:

```
SELECT SQRT(25) "Square Root" FROM DUAL;
```

Output:

```
Square Root
-----------
          5
```

EXP: Returns **e** raised to the **nth** power, where **e** = **2.71828183**.

Syntax:

```
EXP(n)
```

Example:

```
SELECT EXP(5) "Exponent" FROM DUAL;
```

Output:

```
   Exponent
-----------
148.413159
```

EXTRACT: Returns a value extracted from a date or an interval value. A DATE can be used only to extract YEAR, MONTH, and DAY, while a timestamp with a time zone datatype can be used only to extract TIMEZONE_HOUR and TIMEZONE_MINUTE.

Syntax:

```
EXTRACT( {year | month | day | hour | minute | second | timezone_hour |
          timezone_minute | timezone_region | timezone_abbr}
       FROM { date_value | interval_value } )
```

Example:

```
SELECT EXTRACT(YEAR FROM DATE '2004-07-02') "Year",
   EXTRACT(MONTH FROM SYSDATE) "Month" FROM DUAL;
```

Output:

```
Year  Month
----  -----
2004      7
```

GREATEST: Returns the greatest value in a list of expressions.

Syntax:

```
GREATEST(expr1, expr2, ... expr_n)
```

where, **expr1**, **expr2**, ... **expr_n** are expressions that are evaluated by the greatest function.

Example:

```
SELECT GREATEST(4, 5, 17) "Num", GREATEST('4', '5', '17') "Text" FROM DUAL;
```

Output:

```
 Num  Text
----  ----
  17     5
```


LEAST: Returns the least value in a list of expressions.

Syntax:

```
LEAST(expr1, expr2, ... expr_n)
```

where, **expr1**, **expr2**, ... **expr_n** are expressions that are evaluated by the least function.

Example:

```
SELECT LEAST(4, 5, 17) "Num", LEAST('4', '5', '17') "Text" FROM DUAL;
```

Output:

```
 Num  Text
----  ----
   4    17
```

Note

In the **GREATEST()** and **LEAST()** function if the datatypes of the expressions are different, all expressions will be converted to whatever is datatype of the first expression in the list. If the comparison is based on a character comparison, one character is considered greater than another if it has a higher character set value.

MOD: Returns the remainder of a first number divided by second number passed a parameter. If the second number is zero, the result is the same as the first number.

Syntax:

```
MOD(m, n)
```

Example:

```
SELECT MOD(15, 7) "Mod1", MOD(15.7, 7) "Mod2" FROM DUAL;
```

Output:

```
 Mod1  Mod2
----  ----
   1   1.7
```

TRUNC: Returns a number truncated to a certain number of decimal places. The decimal place value must be an integer. If this parameter is omitted, the TRUNC function will truncate the number to 0 decimal places.

Syntax:

```
TRUNC(number, [decimal_places])
```

Example:

```
SELECT TRUNC(125.815, 1) "Trunc1", TRUNC(125.815, -2) "Trunc2" FROM DUAL;
```

Output:

```
 Trunc1  Trunc2
-------  ------
  125.8     100
```

FLOOR: Returns the largest integer value that is equal to or less than a number.

Syntax:

```
FLOOR(n)
```

Example:

```
SELECT FLOOR(24.8) "Flr1", FLOOR(13.15) "Flr2" FROM DUAL;
```

Output:

```
 Flr1  Flr2
   24    13
```

CEIL: Returns the smallest integer value that is greater than or equal to a number.

Syntax:

```
CEIL(n)
```

Example:
SELECT CEIL(24.8) "Ceil1",**CEIL**(13.15) "Ceil2" **FROM DUAL;**

Output:

```
Ceil1 Ceil2
   25    14
```

Note

Several other Numeric functions are available in Oracle. These include the following:

- **ACOS(), ASIN(), ATAN(), ATAN2(),**
- **COS(), COSH(), SIN(), SINH(), TAN(), TANH(),**
- **COVAR_POP(), COVAR_SAMP(), VAR_POP(), VAR_SAMP(),**
- **CORR(), SIGN()**

String Functions

LOWER: Returns char, with all letters in lowercase.

Syntax:

```
LOWER(char)
```

Example:
SELECT LOWER('IVAN BAYROSS') "Lower" **FROM DUAL;**

Output:

```
Lower
-------------
ivan bayross
```

INITCAP: Returns a string with the first letter of each word in **upper case.**

Syntax:

```
INITCAP(char)
```

Example:
SELECT INITCAP('IVAN BAYROSS') "Title Case" **FROM DUAL;**

Output:

```
Title Case
----------
Ivan Bayross
```

UPPER: Returns char, with all letters forced to uppercase.

Syntax:

```
UPPER (char)
```


Example:
SELECT UPPER('Ms. Carol') "Capitalised" **FROM DUAL;**

Output:

```
Capitalised
-----------
MS. CAROL
```

SUBSTR: Returns a portion of characters, beginning at character **m**, and going upto character **n.** If **n** is omitted, the result returned is upto the last character in the string. The first position of char is **1**.

Syntax:

```
SUBSTR(<string>, <start_position>, [<length>])
```

where, **string** is the source string.
start_position is the position for extraction. The first position in the string is always 1.
length is the number of characters to extract.

Example:
SELECT SUBSTR('SECURE',3,4) "Substring" **FROM DUAL;**

Output:

```
Substring
---------
CURE
```

ASCII: Returns the NUMBER code that represents the specified character. If more than one character is entered, the function will return the value for the first character and ignore all of the characters after the first.

Syntax:

```
ASCII(<single_character>)
```

where, **single_character** is the specified character to retrieve the NUMBER code for.

Example:
SELECT ASCII('a') "ASCII1", **ASCII**('A') "ASCII2" **FROM DUAL;**

Output:

```
ASCII1  ASCII2
    97      65
```

COMPOSE: Returns a Unicode string. It can be a **char, varchar2, nchar, nvarchar2, clob,** or **nclob.**

Syntax:

```
COMPOSE(<single>)
```

Below is a listing of **unistring** values that can be combined with other characters in the compose function.

Unistring Value	Resulting character
UNISTR('\0300')	grave accent (`)
UNISTR('\0301')	acute accent (')
UNISTR('\0302')	circumflex (^)
UNISTR('\0303')	tilde (~)
UNISTR('\0308')	umlaut (¨)

Example:
SELECT COMPOSE('a' || UNISTR('\0301')) "Composed" FROM DUAL;

Output:

```
Composed
á
```

DECOMPOSE: Accepts a string and returns a Unicode string.

Syntax:

```
DECOMPOSE(<single>)
```

Example:

```
SELECT DECOMPOSE(COMPOSE('a' || UNISTR('\0301'))) "Decomposed" FROM DUAL;
```

Output:

```
Decomposed
a
```

INSTR: Returns the location of a substring in a string.

Syntax:

```
INSTR(<string1>, <string2>, [<start_position>], [<nth_appearance>])
```

where, **string1** is the string to search.
string2 is the substring to search for in string1.
start_position is the position in **string1** where the search will start. If omitted, it defaults to 1. The first position in the string is 1. If the **start_position** is negative, the function counts back **start_position** number of characters from the end of **string1** and then searches towards the beginning of **string1**.
nth_appearance is the **nth** appearance of **string2**. If omiited, it defaults to 1.

Example:

```
SELECT INSTR('SCT on the net', 't') "Instr1", INSTR('SCT on the net', 't', 1, 2) "Instr2"
  FROM DUAL;
```

Output:

```
 Instr1  Instr2
      8      14
```

TRANSLATE: Replaces a sequence of characters in a string with another set of characters. However, it replaces a single character at a time. For example, it will replace the 1st character in the string_to_replace with the 1st character in the replacement_string. Then it will replace the 2nd character in the string_to_replace with the 2nd character in the replacement_string, and so on.

Syntax:

```
TRANSLATE(<string1>, <string_to_replace>, <replacement_string>)
```

where, **string1** is the string to replace a sequence of characters with another set of characters.
string_to_replace is the string that will be searched for in string1.
All characters in the **string_to_replace** will be replaced with the corresponding character in the **replacement_string**.

Example:

```
SELECT TRANSLATE('1sct523', '123', '7a9') "Change" FROM DUAL;
```

Output:

```
Change
7sct5a9
```


LENGTH: Returns the length of a word.

Syntax:

```
LENGTH(word)
```

Example:

```
SELECT LENGTH('SHARANAM') "Length" FROM DUAL;
```

Output:

```
Length
     8
```

LTRIM: Removes characters from the left of char with initial characters removed upto the first character not in set.

Syntax:

```
LTRIM(char[,set])
```

Example:

```
SELECT LTRIM('NISHA','N') "LTRIM" FROM DUAL;
```

Output:

```
LTRIM
ISHA
```

RTRIM: Returns char, with final characters removed after the last character not in the set. **'set'** is optional, it defaults to spaces.

Syntax:

```
RTRIM (char,[set])
```

Example:

```
SELECT RTRIM('SUNILA','A') "RTRIM" FROM DUAL;
```

Output:

```
RTRIM
SUNIL
```

TRIM: Removes all specified characters either from the beginning or the ending of a string.

Syntax:

```
TRIM( [leading | trailing | both [<trim_character> FROM ] ]  <string1> )
```

where, **leading** - remove **trim_string** from the front of **string1**.
trailing - remove **trim_string** from the end of **string1**.
both - remove **trim_string** from the front and end of **string1**.
If none of the above option is chosen, the **TRIM** function will remove **trim_string** from both the front and end of **string1**.
trim_character is the character that will be removed from string1. If this parameter is omitted, the trim function will remove all leading and trailing spaces from string1.
string1 is the string to trim.

Example 1:

```
SELECT TRIM('  Hansel  ') "Trim both sides" FROM DUAL;
```

Output:

```
Trim both sides
Hansel
```

Example 2:
SELECT TRIM(LEADING 'x' **FROM** 'xxxHanselxxx') "Remove prefixes" **FROM DUAL;**

Output:

```
Remove prefixes
Hanselxxx
```

Example 3:
SELECT TRIM(BOTH 'x' **FROM** 'xxxHanselxxx') "Remove prefixes N suffixes" **FROM DUAL;**

Output:

```
Remove prefixes N suffixes
Hansel
```

Example 4:
SELECT TRIM(BOTH '1' **FROM** '123Hansel12111') "Remove string" **FROM DUAL;**

Output:

```
Remove string
23Hansel12
```

LPAD: Returns **char1**, left-padded to length **n** with the sequence of characters specified in **char2**. If **char2** is not specified Oracle uses blanks by default.

Syntax:

LPAD(char1,n [,char2])

Example:
SELECT LPAD('Page 1',10,'*') "LPAD" **FROM DUAL;**

Output:

```
LPAD
----------
****Page1
```

RPAD: Returns **char1**, right-padded to length **n** with the characters specified in **char2**. If **char2** is not specified, Oracle uses blanks by default.

Syntax:

RPAD(char1,n[,char2])

Example:
SELECT RPAD(FNAME,10,'x') "RPAD Example" **FROM** CUST_MSTR
WHERE FNAME = 'Ivan';

Output:

```
RPAD Example
Ivanxxxxxx
```

VSIZE: Returns the number of bytes in the internal representation of an expression.

Syntax:

VSIZE(<expression>)

Example:
SELECT VSIZE('SCT on the net') "Size" **FROM DUAL;**

Output:

```
Size
----------
14
```

Conversion Functions

TO_NUMBER: Converts **char**, a **CHARACTER** value expressing a number, to a NUMBER datatype.

Syntax:

TO_NUMBER(char)

Example:
UPDATE ACCT_MSTR **SET** Curbal = Curbal + **TO_NUMBER(SUBSTR**('$100',2,3));

Output:

```
10 rows updated.
```

Note

Here, the value 100 will be added to every accounts current balance in the Acct_Mstr table.

TO_CHAR (number conversion): Converts a value of a **NUMBER** datatype to a **character** datatype, using the optional format string. TO_CHAR() accepts a number (**n**) and a numeric format (**fmt**) in which the number has to appear. If **fmt** is omitted, **n** is converted to a char value exactly long enough to hold all significant digits.

Syntax:

TO_CHAR (n[,fmt])

Example:
SELECT TO_CHAR(17145, '$099,999') "Char" **FROM DUAL;**

Output:

```
Char
---------
$017,145
```

TO_CHAR (date conversion): Converts a value of a **DATE** datatype to **CHAR** value. TO_CHAR() accepts a date, as well as the format (fmt) in which the date has to appear. **fmt** must be a date format. If **fmt** is omitted, the **date** is converted to a character value using the default date format, **i.e.** "DD-MON-YY".

Syntax:

TO_CHAR(date[,fmt])

Example:
SELECT TO_CHAR(DT, 'Month DD, YYYY') "New Date Format" **FROM** Trans_Mstr
WHERE Trans_No = 'T1';

Output:

```
New Date Format
-----------------
January  05, 2003
```

DATE CONVERSION FUNCTIONS

The DATE data type is used to store date and time information. The DATE data type has special properties associated with it. It stores information about century, year, month, day, hour, minute and second for each date value.

The value in the column of a DATE data type, **is always** stored in a specific **default** format. This default format is 'DD-MON-YY HH:MI:SS'. Hence, when a date has to be inserted in a date field, its value has to be specified in the same format. Additionally, values of DATE columns are always displayed in the **default** format when **retrieved** from the table.

If data from a date column has to be viewed in any other format other than the default format, Oracle provides the **TO_DATE** function that can be used to specify the required format.

The same function can also be used for storing a date into a DATE field in a particular format (other than default). This can be done by specifying the date value, along with the format in which it is to be inserted. The TO_DATE() function also allows part insertion of a DATE value into a column, for example, only the day and month portion of the date value.

To enter the time portion of a date, the TO_DATE function must be used with a **format mask** indicating the time portion.

TO_DATE: Converts a character field to a date field.

Syntax:

```
TO_DATE(char [, fmt])
```

Example:

```
INSERT INTO CUST_MSTR(CUST_NO, FNAME, MNAME, LNAME, DOB_INC)
    VALUES('C1', 'Ivan', 'Nelson', 'Bayross',
        TO_DATE('25-JUN-1952 10:55 A.M.', 'DD-MON-YY HH:MI  A.M.'));
```

Output:

```
1 rows created.
```

DATE FUNCTIONS

To manipulate and extract values from the date column of a table Oracle provides some date functions. These are discussed below:

ADD_MONTHS: Returns date after adding the number of months specified in the function.

Syntax:

```
ADD_MONTHS(d,n)
```

Example:

```
SELECT ADD_MONTHS(SYSDATE, 4) "Add Months" FROM DUAL;
```

Output:

```
Add Months
----------
01-NOV-04
```

LAST_DAY: Returns the last date of the month specified with the function.

Syntax:

```
LAST_DAY(d)
```

Example:

```
SELECT SYSDATE, LAST_DAY(SYSDATE) "LastDay" FROM DUAL;
```

Output:

```
SYSDATE   LastDay
--------- ---------
01-JUL-04 31-JUL-04
```

MONTHS_BETWEEN: Returns number of months between **d1** and **d2**.

Syntax:

```
MONTHS_BETWEEN(d1, d2)
```

Example:

```
SELECT MONTHS_BETWEEN('02-FEB-92', '02-JAN-92') "Months" FROM DUAL;
```

Output:

```
Months
----------
         1
```

NEXT_DAY: Returns the date of the first weekday named by **char** that is after the date named by **date**. **char** must be a day of the week.

Syntax:

```
NEXT_DAY(date, char)
```

Example:

```
SELECT NEXT_DAY('06-JULY-02', 'Saturday') "NEXT DAY" FROM DUAL;
```

Output:

```
NEXT DAY
---------
13-July-02
```

ROUND: Returns a date rounded to a specific unit of measure. If the second parameter is omitted, the **ROUND** function will round the date to the nearest day.

Syntax:

```
ROUND(date, [format])
```

Below are the valid format parameters:

Unit	Format parameters	Rounding Rule
Year	SYYYY, YYYY, YEAR, SYEAR, YYY, YY, Y	Rounds up on July 1st
ISO Year	IYYY, IY, I	
Quarter	Q	Rounds up on the 16th day of the second month of the quarter
Month	MONTH, MON, MM, RM	Rounds up on the 16th day of the month
Week	WW	Same day of the week as the first day of the year
IW	IW	Same day of the week as the first day of the ISO year
W	W	Same day of the week as the first day of the month
Day	DDD, DD, J	
Hour	HH, HH12, HH24	

Unit	Format parameters	Rounding Rule
Start day of the week	DAY, DY, D	
Minute	MI	

Example:
SELECT ROUND(TO_DATE('01-JUL-04'), 'YYYY') "Year" **FROM DUAL;**

Output:
```
Year
---------
01-JAN-05
```

NEW_TIME: Returns the date after converting it from **time zone1** to a date in **time zone2**.

Syntax:
NEW_TIME(date, zone1, zone2)

Value	Description	Value	Description
AST	Atlantic Standard Time	ADT	Atlantic Daylight Time
BST	Bering Standard Time	BDT	Bering Daylight Time
CST	Central Standard Time	CDT	Central Daylight Time
EST	Eastern Standard Time	EDT	Eastern Daylight Time
GMT	Greenwich Mean Time	HST	Alaska-Hawaii Standard Time
HDT	Alaska-Hawaii Daylight Time	MST	Mountain Standard Time
MDT	Mountain Daylight Time	NST	Newfoundland Standard Time
PST	Pacific Standard Time	PDT	Pacific Daylight Time
YST	Yukon Standard Time	YDT	Yukon Daylight Time

Example:
The following example converts an Atlantic Standard Time into a Mountain Standard Time:
SELECT NEW_TIME(TO_DATE('2004/07/01 01:45', 'yyyy/mm/dd HH24:MI'), 'AST', 'MST') "MST"
FROM DUAL;

Output:
```
MST
---------
30-JUN-04
```

Note

Several other Date function are available in Oracle. These include the following:
- **DbTimeZone(), SessionTimeZone(), SysTimestamp(), Tz_Offset()**

The above Oracle date functions are **just a few** selected from the many date functions that are built into Oracle. These Oracle functions are commonly used in commercial application development.

MANIPULATING DATES IN SQL USING THE DATE()

A column of data type **Date** is always displayed in a default format, which is **'DD-MON-YY'**. If this default format is not used when entering data into a column of the **date** data type, Oracle **rejects the data** and returns an error message.

If a **date** has to be retrieved or inserted into a table in a format **other than** the default one, Oracle provides the **TO_CHAR** and **TO_DATE** functions to do this.

TO_CHAR

The TO_CHAR function facilitates the retrieval of data in a format different from the default format. It can also extract a part of the date, i.e. the date, month, or the year from the date value and use it for sorting or grouping of data according to the date, month, or year.

Syntax:
TO_CHAR(<date value> [,<fmt>])
where **date value** stands for the date and **fmt** is the specified format in which date is to be displayed.

Example 1:
SELECT TO_CHAR(SYSDATE, 'DD-MM-YY') **FROM DUAL;**

Output:
```
TO_CHAR(
---------
01-07-04
```

TO_DATE

TO_DATE converts a **char** value into a **date** value. It allows a user to insert date into a date column in any required format, by specifying the **character** value of the date to be inserted and its format.

Syntax:
TO_DATE(<char value>[,<fmt>])
where **char value** stands for the value to be inserted in the date column, and **fmt** is a date format in which the 'char value' is specified.

Example 2:
SELECT TO_DATE ('06/07/02', 'DD/MM/YY') **FROM DUAL;**

Output:
```
TO_DATE('
---------
06-JUL-02
```

Example 3:
List the transaction details in order of the months for account no. **SB9**. The **Transaction Date** should be displayed in **'DD/MM/YY'** format.

Synopsis:

Tables:	TRANS_MSTR
Columns:	TRANS_NO, ACCT_NO, DT, PARTICULAR, DR_CR, AMT, BALANCE
Technique:	**Functions:** TO_CHAR(), **Clauses:** WHERE, ORDER BY

Solution:
SELECT TRANS_NO, ACCT_NO, TO_CHAR(DT, 'DD/MM/YY') "Transaction Date",
PARTICULAR, DR_CR, AMT, BALANCE
FROM TRANS_MSTR **WHERE** ACCT_NO = 'SB9' **ORDER BY TO_CHAR**(DT, 'MM');

Output:

TRANS_NO	ACCT_NO	Transaction Date	PARTICULAR	DR_CR	AMT	BALANCE
T9	SB9	05/04/03	Initial Payment	D	500	500
T10	SB9	15/04/03	CLR-204907	D	3000	3500
T11	SB9	17/04/03	Self	W	2500	1000
T13	SB9	05/06/03	CLR-204908	D	3000	4000

Output: (Continued)

TRANS_NO	ACCT_NO	Transaction Date	PARTICULAR	DR_CR	AMT	BALANCE
T14	SB9	27/06/03	Self	W	2500	1500

Explanation:
Here the value held in the DT field is formatted using the TO_CHAR() function to display the date in the **DD/MM/YY** format. The ordering of the output data set is based on the "**MONTH**" segment of the data in the column DT. This is done using the **TO_CHAR()** function, in the order by clause, extracting only the "**MONTH**" segment of the DT to sort on.

Example 4:
Insert the following data in the table CUST_MSTR, wherein the time component has to be stored along with the date in the column DOB_INC.

Cust_No	Fname	Lname	Dob_Inc
C100	Sharanam	Shah	03/Jan/1981 12:23:00

```
INSERT INTO CUST_MSTR (CUST_NO, FNAME, LNAME, DOB_INC)
    VALUES('C100', 'Sharanam', 'Shah', TO_DATE('03/Jan/1981 12:23:00', 'DD/MON/YY hh:mi:ss'));
```

Output:

```
1 row created.
```

Special Date Formats Using TO_CHAR function

Sometimes, the date value is required to be displayed in special formats, for example, instead of 03-JAN-81, displays the date as 03rd of January, 1981. For this, Oracle provides **special attributes**, which can be used in the format specified with the **TO_CHAR** and **TO_DATE** functions. The significance and use of these characters are explained in the examples below.

All three examples below are based on the CUST_MSTR table

The query is as follows:

```
SELECT CUST_NO, FNAME, LNAME, DOB_INC
    FROM CUST_MSTR WHERE CUST_NO LIKE 'C_';
```

Output:

CUST_NO	FNAME	LNAME	DOB_INC
C1	Ivan	Bayross	25-JUN-52
C2	Chriselle	Bayross	29-OCT-82
C3	Mamta	Muzumdar	28-AUG-75
C4	Chhaya	Bankar	06-OCT-76
C5	Ashwini	Joshi	20-NOV-78
C6	Hansel	Colaco	01-JAN-82
C7	Anil	Dhone	12-OCT-83
C8	Alex	Fernandes	30-SEP-62
C9	Ashwini	Apte	19-APR-79

9 rows selected.

Variations in this output can be achieved as follows:

- **Use of TH in the TO_CHAR() function:**

DDTH places TH, RD, ND for the date (DD), for example, 2ND, 3RD, 08TH etc

```
SELECT CUST_NO, FNAME, LNAME, TO_CHAR(DOB_INC, 'DDTH-MON-YY') "DOB_DDTH"
    FROM CUST_MSTR WHERE CUST_NO LIKE 'C_';
```


Output:

CUST_NO	FNAME	LNAME	DOB_DDTH
C1	Ivan	Bayross	25TH-JUN-52
C2	Chriselle	Bayross	29TH-OCT-82
C3	Mamta	Muzumdar	28TH-AUG-75
C4	Chhaya	Bankar	06TH-OCT-76
C5	Ashwini	Joshi	20TH-NOV-78
C6	Hansel	Colaco	01ST-JAN-82
C7	Anil	Dhone	12TH-OCT-83
C8	Alex	Fernandes	30TH-SEP-62
C9	Ashwini	Apte	19TH-APR-79

9 rows selected.

- **Use of SP in the TO_CHAR() function**

DDSP indicates that the date (DD) must be displayed by spelling the date such as ONE, TWELVE etc.

```
SELECT CUST_NO, FNAME, LNAME, TO_CHAR(DOB_INC, 'DDSP') "DOB_DDSP"
    FROM CUST_MSTR WHERE CUST_NO LIKE 'C_';
```

Output:

CUST_NO	FNAME	LNAME	DOB_DDSP
C1	Ivan	Bayross	TWENTY-FIVE
C2	Chriselle	Bayross	TWENTY-NINE
C3	Mamta	Muzumdar	TWENTY-EIGHT
C4	Chhaya	Bankar	SIX
C5	Ashwini	Joshi	TWENTY
C6	Hansel	Colaco	ONE
C7	Anil	Dhone	TWELVE
C8	Alex	Fernandes	THIRTY
C9	Ashwini	Apte	NINETEEN

9 rows selected.

- **Use of 'SPTH' in the to_char function**

SPTH displays the date (DD) with **th** added to the spelling fourteen**th**, twelf**th**.

```
SELECT CUST_NO, FNAME, LNAME, TO_CHAR(DOB_INC, 'DDSPTH') "DOB_DDSPTH"
    FROM CUST_MSTR WHERE CUST_NO LIKE 'C_';
```

Output:

CUST_NO	FNAME	LNAME	DOB_DDSPTH
C1	Ivan	Bayross	TWENTY-FIFTH
C2	Chriselle	Bayross	TWENTY-NINTH
C3	Mamta	Muzumdar	TWENTY-EIGHTH
C4	Chhaya	Bankar	SIXTH
C5	Ashwini	Joshi	TWENTIETH
C6	Hansel	Colaco	FIRST
C7	Anil	Dhone	TWELFTH
C8	Alex	Fernandes	THIRTIETH
C9	Ashwini	Apte	NINETEENTH

9 rows selected.

MISCELLANEOUS FUNCTIONS

UID: This function returns an integer value corresponding to the UserID of the user currently logged in.

Syntax:

```
UID [INTO <variable>]
```

where, **variable** will now contain the id number for the user's session.

Example:

```
SELECT UID FROM DUAL;
```

Output:

```
   UID
------
    61
```

USER: This function returns the **user name** of the user who has logged in. The value returned is in varchar2 data type.

Syntax:

```
USER
```

Example:

```
SELECT USER FROM DUAL;
```

Output:

```
USER
------------------------------
DBA_BANKSYS
```

SYS_CONTEXT: Can be used to retrieve information about Oracle's environment.

Syntax:

```
SYS_CONTEXT (<namespace>, <parameter>, [<length>])
```

where, **namespace** is an Oracle namespace that has already been created. If the **namespace** of **USERENV** is used, attributes describing the current Oracle session can be returned.
parameter is a valid attribute that has been set using the DBMS_SESSION.set_context procedure.
length is the length of the return value in bytes. If this parameter is omitted or if an invalid entry is provided, the **SYS_CONTEXT** function will default to **256 bytes**.

The valid parameters for the namespace called **USERENV** are as follows:

Parameter	Explanation	Return Length
AUDITED_CURSORID	Returns the cursor ID of the SQL that triggered the audit	N/A
AUTHENTICATION_DATA	Authentication data	256
AUTHENTICATION_TYPE	Describes how the user was authenticated. Can be one of the following values: Database, OS, Network, or Proxy	30
BG_JOB_ID	If the session was established by an Oracle background process, this parameter will return the Job ID. Otherwise, it will return NULL.	30
CLIENT_IDENTIFIER	Returns the client identifier (global context)	64
CLIENT_INFO	User session information	64
CURRENT_SCHEMA	Returns the default schema used in the current schema	30
CURRENT_SQL	Returns the SQL that triggered the audit event	64
CURRENT_USER	Name of the current user	30
CURRENT_USERID	Userid of the current user	30
DB_NAME	Name of the database from the DB_NAME initialization parameter	30
ENTRYID	Available auditing entry identifier	30
EXTERNAL_NAME	External of the database user	256
HOST	Name of the host machine from which the client has connected	54

Parameter	Explanation	Return Length
CURRENT_SCHEMAID	Returns the identifier of the default schema used in the current schema	30
DB_DOMAIN	Domain of the database from the DB_DOMAIN initialization parameter	256
FG_JOB_ID	If the session was established by a client foreground process, this parameter will return the Job ID. Otherwise, it will return NULL.	30
GLOBAL_CONTEXT_MEMORY	The number used in the System Global Area by the globally accessed context	N/A
INSTANCE	The identifier number of the current instance	30
IP_ADDRESS	IP address of the machine from which the client has connected	30
ISDBA	Returns TRUE if the user has DBA privileges. Otherwise, it will return FALSE.	30
LANG	The ISO abbreviate for the language	62
LANGUAGE	The language, territory, and character of the session. In the following format: language_territory.characterset	52
NETWORK_PROTOCOL	Network protocol used	256
NLS_CALENDAR	The calendar of the current session	62
NLS_CURRENCY	The currency of the current session	62
NLS_DATE_FORMAT	The date format for the current session	62
NLS_DATE_LANGUAGE	The language used for dates	62
NLS_SORT	BINARY or the linguistic sort basis	62
NLS_TERRITORY	The territory of the current session	62
OS_USER	The OS username for the user logged in	30
PROXY_USER	The name of the user who opened the current session on behalf of SESSION_USER	30
PROXY_USERID	The identifier of the user who opened the current session on behalf of SESSION_USER	30
SESSION_USER	The database user name of the user logged in	30
SESSION_USERID	The database identifier of the user logged in	30
SESSIONID	The identifier of the auditing session	30
TERMINAL	The OS identifier of the current session	10

Example:

```
SELECT SYS_CONTEXT('USERENV', 'NLS_DATE_FORMAT') "SysContext" FROM DUAL;
```

Output:

```
SysContext
----------
DD-MON-RR
```

USERENV: Can be used to retrieve information about the current Oracle session. Although this function still exists in Oracle for backwards compatibility, it is recommended that the **SYS_CONTEXT** function is used instead.

Syntax:

```
USERENV(<parameter>)
```

where, **parameter** is the value to return from the current Oracle session.

The possible values are:

Parameter	Explanation
CLIENT_INFO	Returns user session information stored using the DBMS_APPLICATION_INFO package
ENTRYID	Available auditing entry identifier

INSTANCE	The identifier number of the current instance
ISDBA	Returns TRUE if the user has DBA privileges. Otherwise, it will return FALSE.
LANG	The ISO abbreviate for the language
LANGUAGE	The language, territory, and character of the session. In the following format: language_territory.characterset
SESSIONID	The identifier of the auditing session
TERMINAL	The OS identifier of the current session

Example:
SELECT USERENV('LANGUAGE') **FROM DUAL;**

Output:
```
USERENV('LANGUAGE')
----------------------------------------
AMERICAN_AMERICA.WE8MSWIN1252
```

COALESCE: Returns the first non-null expression in the list. If all expressions evaluate to null, then the **coalesce** function will return null.

Syntax:

COALESCE(<expr1>, <expr2>, ... <expr_n>)

Example:
SELECT COALESCE(FNAME, CUST_NO) **Customers FROM** CUST_MSTR;

The above coalesce statement is equivalent to the following IF-THEN-ELSE statement:

```
IF FNAME IS NOT NULL THEN
    Customers := FNAME;
ELSIF CUST_NO IS NOT NULL THEN
    Customers := CUST_NO;
ELSE
    Customers := NULL;
END IF;
```

Output:
```
CUSTOMERS
----------------------------------------
Ivan
Chriselle
Mamta
Chhaya
Ashwini
Hansel
Anil
Alex
Ashwini
Namita
O11
O12
O13
O14
```

Explanation:
In the above example, Oracle will display the first name i.e. the value held in the field FNAME if first name field holds a value. If does not hold a value, then Oracle will move on to the next column in the **COALESCE** function and display the value held in the next column i.e. CUST_NO if it hold a value.

In case the second column also does not hold a value, then Oracle will display null as an output.

SELF REVIEW QUESTIONS

FILL IN THE BLANKS

1. The Oracle engine will process all rows in a table and display the result only when any of the conditions specified using the _____ operator are satisfied.
2. The _____ predicate allows for a comparison of one string value with another string value, which is not identical.
3. For character datatypes the _____ sign matches any string.
4. _____ is a small Oracle worktable, which consists of only one row and one column, and contains the value x in that column.
5. Functions that act on a set of values are called as _____ _____.
6. Variables or constants accepting by functions are called _____.
7. The _____ function returns a string with the first letter of each word in upper case.
8. The _____ function removes characters from the left of char with initial characters removed upto the first character not in set.
9. _____ returns the string passed as a parameter after right padding it to a specified length.
10. The _____ function converts char, a CHARACTER value expressing a number, to a NUMBER datatype.
11. The _____ function converts a value of a DATE datatype to CHAR value.
12. The _____ function returns number of months between two dates.
13. The _____ function returns an integer value corresponding to the UserID of the user currently logged in.

TRUE OR FALSE

14. The Oracle engine will process all rows in a table and display the result only when none of the conditions specified using the NOT operator are satisfied.
15. In order to select data that is within a range of values, the IN BETWEEN operator is used.
16. For character datatypes the percent sign matches any single character.
17. COUNT(expr) function returns the number of rows where expr is not null.
18. ROOT function returns square root of a numeric value.
19. The second parameter in the ROUND function specifies the number of digits after the decimal point.
20. The LOWER function returns char, with all letters in lowercase.
21. The UPPER function returns a string with the first letter of each word in upper case.
22. The LENGTH function returns the length of a word.
23. The LTRIM returns char, with final characters removed after the last character not in the set. 'set' is optional, it defaults to spaces.

24. LPAD returns the string passed as a parameter after left padding it to a specified length.
25. The TO_CHAR (date conversion) converts a value of a NUMBER datatype to a character datatype, using the optional format string.
26. The DATE data type is used to store date and time information.
27. The TO_DATE() function also disallows part insertion of a DATE value into a column.
28. The ADD_MONTHS function returns date after adding the number of months specified in the function.
29. The TO-DATE function allows a user to insert date into a date column in any required format, by specifying the character value of the date to be inserted and its format.

HANDS ON EXERCISES

Using the tables created previously generate the SQL statements for the operations mentioned below. The tables in user are as follows:

a. Client_Master
b. Product_Master
c. Salesman_Master
d. Sales_Order
e. Sales_Order_Details

1. Perform the following computations on table data:

a. List the names of all clients having 'a' as the second letter in their names.
b. List the clients who stay in a city whose First letter is 'M'.
c. List all clients who stay in 'Bangalore' or 'Mangalore'
d. List all clients whose BalDue is greater than value 10000.
e. List all information from the Sales_Order table for orders placed in the month of June.
f. List the order information for ClientNo 'C00001' and 'C00002'.
g. List products whose selling price is greater than 500 and less than or equal to 750.
h. List products whose selling price is more than 500. Calculate a new selling price as, original selling price * .15. Rename the new column in the output of the above query as new_price.
i. List the names, city and state of clients who are not in the state of 'Maharashtra'.
j. Count the total number of orders.
k. Calculate the average price of all the products.
l. Determine the maximum and minimum product prices. Rename the output as max_price and min_price respectively.
m. Count the number of products having price less than or equal to 500.
n. List all the products whose QtyOnHand is less than reorder level.

2. Exercise on Date Manipulation:

a. List the order number and day on which clients placed their order.
b. List the month (in alphabets) and date when the orders must be delivered.
c. List the OrderDate in the format 'DD-Month-YY'. e.g. 12-February-02.
d. List the date, 15 days after today's date.

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	CA	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 filte. 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 groupings 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	SUM(CURBAL)
	SB15	500
B1		**23000**
B1	CA7	22000
B1	SB1	500
B1	SB11	500
B2		**8500**
B2	CA2	3000
B2	SB8	500
B2	CA12	5000
B3		1000
B3	SB3	500
B3	SB13	500
B4		**1000**
B4	SB6	500
B4	SB9	500
B5		**22000**
B5	CA4	12000
B5	CA14	10000
B6		**33000**
B6	SB5	500
B6	CA10	32000
B6	SB15	500

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 that 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
CA10       19-FEB-04   D      Cheque    2000
SB9        05-APR-04   D      Cheque    3000
SB9        27-APR-04   W      Cash      2500
SB11       05-MAR-04   D      Cash       500
CA12       10-MAR-04   D      Cash      2000
SB13       22-MAR-04   D      Cash       500
CA14       05-APR-04   D      Cheque    2000
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
E1	Ivan	Bayross
E4	Peter	Joseph

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>, <ColumnName N>
    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.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_NO Name                 Branch          DEPT                DESIG
------ -------------------- --------------- ------------------- -------------------
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	500
C10	Namita S. Kanade	CA10	Darya Ganj	32000
C10	Namita S. Kanade	SB9	Mahim	500
C2	Chriselle Ivan Bayross	CA12	Andheri	5000
C2	Chriselle Ivan Bayross	CA2	Andheri	3000
C3	Mamta Arvind Muzumdar	CA12	Andheri	5000
C3	Mamta Arvind Muzumdar	CA2	Andheri	3000
C3	Mamta Arvind Muzumdar	SB9	Mahim	500
C4	Chhaya Sudhakar Bankar	CA14	Borivali	10000
C4	Chhaya Sudhakar Bankar	CA4	Borivali	12000
C4	Chhaya Sudhakar Bankar	SB13	Churchgate	500
C4	Chhaya Sudhakar Bankar	SB15	Darya Ganj	500
C4	Chhaya Sudhakar Bankar	SB3	Churchgate	500
C4	Chhaya Sudhakar Bankar	SB5	Darya Ganj	500
C5	Ashwini Dilip Joshi	CA14	Borivali	10000
C5	Ashwini Dilip Joshi	CA4	Borivali	12000
C5	Ashwini Dilip Joshi	SB6	Mahim	500
C6	Hansel I. Colaco	CA7	Vile Parle (HO)	22000
C7	Anil Arun Dhone	SB6	Mahim	500
C8	Alex Austin Fernandes	CA7	Vile Parle (HO)	22000
C9	Ashwini Shankar Apte	CA10	Darya Ganj	32000
C9	Ashwini Shankar Apte	SB8	Andheri	500

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                 CNTC_TYPE  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        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.

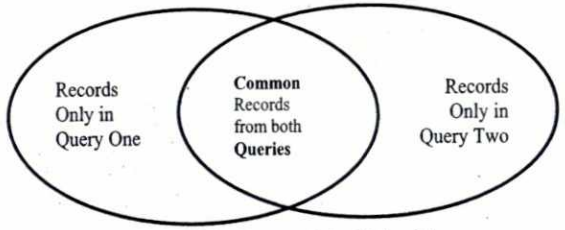

Diagram 10.1: Output of the Union Clause.

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.

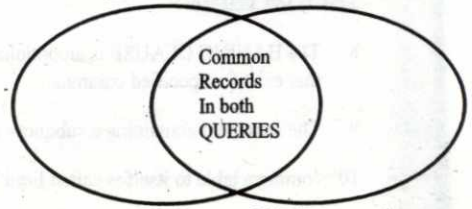

Diagram 10.2: Output of the Intersect clause.

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 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.

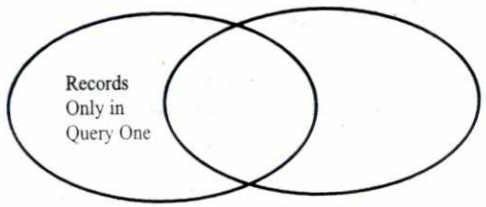

Diagram 10.3: Output of the Minus clause.

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
```

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 and 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 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.

ANSWERS TO SELF REVIEW QUESTIONS

7. INTERACTIVE SQL: PART – I

FILL IN THE BLANKS

1. table
2. Create Table
3. single
4. Where Clause
5. SELECT DISTINCT
6. Target, Source
7. ALTER TABLE
8. UPDATE
9. single record

TRUE OR FALSE

10. False
11. False
12. True
13. False
14. True
15. False
16. True
17. True
18. False
19. True

8. INTERACTIVE SQL: PART – II

FILL IN THE BLANKS

1. Constraints
2. NULLABLE
3. mandatory
4. NOT NULL
5. NULL
6. Simple
7. UNIQUE
8. Foreign
9. Detail
10. ON DELETE CASCADE
11. CHECK
12. Boolean
13. Integrity
14. indexes

TRUE OR FALSE

15. True
16. False
17. False
18. True
19. False
20. True
21. False
22. True
23. True
24. False
25. True
26. False
27. True
28. True