COMPUTER SCIENCE

B. Sc. II (CBCS)
Semester-IV
2023-2024

2CS2 : RDBMS and Core Java

Unit-II : Introduction to SQL

Prof. V. V. Agarkar
Assistant Professor & Head
Department of Computer Science

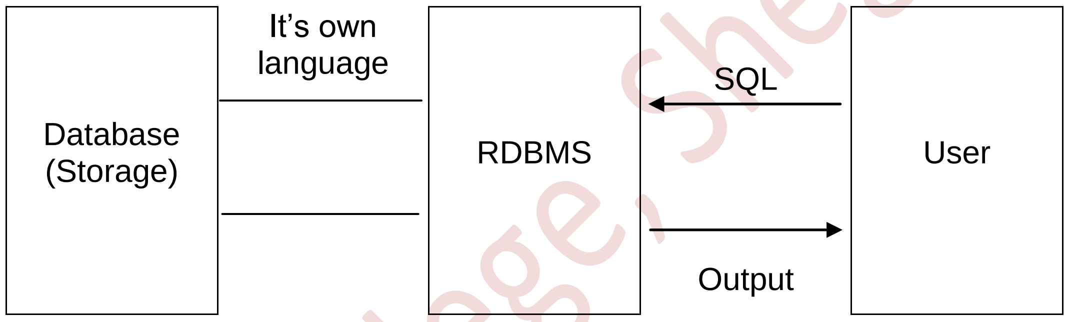
Shri. D. M. Burungale Science & Arts College, Shegaon, Dist. Buldana

UNIT - II

***Syllabus*: Introduction to SQL**: Components of SQL, data types, operators, DDL Commands: CREATE, ALTER, DROP, RENAME, DML Commands: SELECT, INSERT, DELETE & UPDATE; Clauses: ORDER BY, GROUP BY and HAVING; Joins and Unions: Self, Equi and Outer Join, Unions and Intersection. Functions: aggregate functions, string functions.

Introduction

RDBMS is mainly used for management of data. The common operations on data are storing the data, manipulating the data and retrieving the data. In multi-user environment it performs additional operations like handling security and concurrency. RDBMS doing these jobs on user request. Generally RDBMS provides a tool for the communication between user and RDBMS, and this tool is SQL. Conceptually SQL can be shown as follows:



[**Fig.1 :** SQL a communication tool between RDBMS and User]

Form this figure-1 it is clear that if user wants to do certain operation he will request to the RDBMS using SQL.

Structured Query Language (SQL)

SQL stands for Structured Query Language. SQL is used to communicate with a database. According to ANSI, it is the standard language for relational database management systems. SQL statements are used to perform tasks such as creation of tables, inserting data, updating data, deleting data and retrieval of data. In addition to this, in multi-user database, SQL provides adequate security to the databases.

Some common relational database management systems that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access, Ingress, etc. Although most database systems use SQL, but the syntax of SQL changes very little from one RDBMS to another. Thus SQL is a ***standard language*** for RDBMS.

The important feature of SQL is that it is ***non-procedural language***. In non-procedural language you have to describe what to do rather than how it is done.

SQL does not have any procedural capabilities such as looping and branching nor does it have any conditional checking capabilities.

Oracle provides an interactive SQL tool *i.e.* **SQL* Plus**, which allows users to enter the ANSI SQL sentences and pass them for execution.

Using SQL user can do:

- Creation of tables (i.e. databases)
- Insertion of data in the tables.
- Manipulation of data in the tables.
- Retrieving data from the tables.
- Deleting the data from the table.
- Controlling the user access to the table for security point of view.

Characteristics / Advantages of SQL

Following are some characteristics / advantages of SQL :

1. SQL is a standardized language of RDBMS. SQL can be used with most of the RDBMS as their language.
2. SQL is a non-procedural language. Thus, in SQL you have to just describe what to do rather than how it is done.
3. SQL is very easy to learn.
4. The embedded form of SQL is designed for the use within general-purpose programming languages, such as PL/I, COBOL, C, Pascal, and Fortran etc.
5. Since SQL is a non-procedural language, using a single line command, we can perform operations on set of records. Thus it reduces the code.
6. SQL commands are self-explanatory English words, thus simplicity in writing code and also ease in writing code.

Components of SQL OR SQL Basic Statements OR Types of SQL statements

The SQL statements are grouped into three different components:

1) Data Definition Language (DDL)
2) Data Manipulation Language (DML)
3) Data Control Language (DCL)

1) Data Definition Language (DDL)

The Data Definition Language (DDL) contains the subset of SQL commands used to modify the actual ***structure*** of a database, rather than the database's contents (data). Using DDL statements, you can:

- Define & create a new table
- Remove a table that's no longer needed
- Change the definition of an existing table (adding or removing columns)
- Define a virtual table (view) of data
- Establish security controls for a database.

Some DDL command is: CREATE TABLE, ALTER TABLE, DROP TABLE etc.

2) Data Manipulation Language (DML)

The Data Manipulation Language (DML) contains the subset of SQL commands used to modify the actual ***contents (data)*** stored in a database, rather than its structure. Using DML Statements, you can

- Insert data into a table
- Update existing data into a table
- Delete data from a table
- Retrieve data from tables.
- Transaction processing

Some DML commands are: INSERT, UPDATE, DELETE, SELECT etc.

3) Data Control Language (DCL)

The Data Control Language (DCL) contains the subset of SQL commands used to manage user access to databases in multi-user environment. It consists of two commands: the GRANT command, used to add database permissions for a user, and the REVOKE command, used to take away existing permissions.

Data Types

When user create a table (database) using SQL, he must have to define the type of data that each column or field will contain. This is done by assigning each column or field a 'data type' that indicates the kind of value the field will contain. Once a data type is assigned to a column, it becomes the column type, or field type, for that column. Following are data types that are allow to use in SQL.

1) CHAR(size)

The CHAR data type accepts character strings. It is used to store alphanumeric characters of ***fixed length***. The CHAR type consists of all the printable characters, including the numbers. The 'size' in brackets determines the maximum number of characters that can be entered into that field. The maximum number of characters this data type can hold is 255 characters (i.e. size is 255 bytes in Oracle 7 and 2000 bytes in Oracle 8, 9i, 10g, and 11g).

If a value that is inserted in a field of CHAR data type is shorter than the size is defined for it, then it will be padded (fill) with spaces on the right until it reaches the size characters in length. Attempting to assign a value containing more characters than the defined length, results in the truncation of the character string to the defined length.

2) VARCHAR2(size)

The VARCHAR2 data type accepts character strings, of a ***variable length***. It is also used to store alphanumeric characters. The maximum size of this data type is 2000 bytes in Oracle 7 and 4000 bytes in Oracle 8, 9i, 10g, and 11g.

If a value that is inserted in a field of VARCHAR2 data type is shorter than the size it is defined for, then it will not be padded with spaces. Attempting to assign a value containing more characters than the defined maximum length, results in the truncation of the character string to the defined length.

3) NUMBER(P, S)

This data type is used to store negative, positive, integer, fixed-decimal and floating point numbers. The precision (P) and scale (S) can be specified for this. The ***precision P*** is a positive integer that indicates the total number of digits in the number, both to the left and to the right of the decimal point. The ***scale S*** is a total number of digits to the right of the decimal point. The precision P can range from 1 to 38.

NUMBER data type can be declared in one of three different ways:

`NUMBER`	– Stores floating point decimal values.
`NUMBER(p)`	– Scale defaults to 0, stores only integer numbers.
`NUMBER(p,s)`	– Precision and Scale are defined by the user and stores fixed-point decimal values.

Examples:

`MARK NUMBER`	The MARK field stores floating point decimal values.
`MARK NUMBER(25)`	The MARK field stores only integer numbers having width 25 digits.
`MARK NUMBER(5,2)`	The MARK field stores fixed-point decimal values it can stores 5 digit numbers having at most 3 digits before decimal point and at most 2 digits after decimal point. It take data in the range of -999.99 to +999.99

4) DECIMAL(P, S)

This data type is used to store negative, positive, integer, fixed-decimal and floating point numbers. When a DECIMAL data type used its precision (P) and scale (S) can be specified. The ***precision P*** is a positive integer that indicates the total number of digits in the number, both to the left and to the right of the decimal point. The ***scale S*** is a total number of digits to the right of the decimal point. The precision P can range from 1 to 38.

5) INTEGER or INT

The INTEGER data type accepts a 32-bit signed integer value with an implied scale of zero. It stores any integer value between the range 2^{-31} and 2^{31}-1. Attempting to assign values outside this range causes an error.

If you assign a numeric value with a precision and scale to an INTEGER data type, the scale portion truncates, without rounding.

6) SMALLINT

The SMALLINT data type accepts a 16-bit signed integer value with an implied scale of zero. It stores any integer value between the range 2^{-15} and 2^{15}-1. Attempting to assign values outside this range causes an error.

If you assign a numeric value with a precision and scale to a SMALLINT data type, the scale portion truncates, without rounding.

7) FLOAT(p)

In float data type, permitted values have a precision but no scale. As a result the decimal point can float. A 'floating-point number' is one that contains a decimal point, but the decimal point can be located at any place within that number, which is why an approximate numeric is said to have no scale. Approximate numeric data types include REAL, DOUBLE PRECISION, and FLOAT.

The FLOAT data type accepts a single or double precision floating point number value, for which you may define a precision up to a maximum of 64. If no precision is specified during the declaration, the default precision is 64. Attempting to assign a value lager than the declared precision will cause an error to be raised.

8) DATE

The DATE data type is used to represent dates. The DATE data type accepts date values, consisting of year, month, and day. No parameters are required when declaring a DATE data type. Date values should be specified in the form: DD-MMM-YY.

You can store any date between JAN 4712 BC to DEC 4711 AD. Month values must be between JAN and DEC, day values should be between 1 and 31 depending on the month. Values assigned to the DATE data type should be enclosed in single quotes.

9) LONG

This data type is used to specify long text. This data type contains variable-length character data up to 4 Giga Bytes (GB). Only one LONG column per table is allowed. To define LONG type, there is no need to specify its size.

10) RAW

This data type is used to store raw binary data, such as graphics. The size is 240 bytes in Oracle 7 and 2000 bytes in Oracle 8, 9i, 10g, and 11g.

11) LONG RAW

It is also used to store raw binary data, such as graphics. But, the size is 4 Giga Bytes (GB).

12) ROWID

Each table in an Oracle database internally has a pseudo-column named ROWID. It is a string representing the unique address of a row in its table.

ROWID data type is primarily used for values returned by the ROWID pseudo-column. Its size is fixed at 10 bytes (extended ROWID) or 6 bytes (restricted ROWID) for each row in the table.

Other advanced data types used in Oracle are: `NCHAR`, `BLOB`, `CLOB`, `NCLOB`, `BFILE`, etc.

OPERATORS

The SQL operators are either special words or characters that tell SQL to execute an operation. These operations can be mathematical calculations, value comparisons or clause connections.

Generally there are three types of operators in SQL:

1) Arithmetic or value operators
2) Relational or comparison operators
3) Logical or Boolean operators.

1) Arithmetic <u>or</u> value operators

Following arithmetic operators are used in SQL:

Operator	Meaning
Unary + or –	for giving sign to number or expression.
+	for Addition
-	for Subtraction
*	for Multiplication
/	for Division

Example:

```
SELECT NAME, PHY, MTH, CPS,(PHY+MTH+CPS);
```

2) Relational / Comparison operator

These operators are used to compare two values or two expressions. The result of the comparison is TRUE or FALSE. Following operators are used as a logical operator.

Operator	Meaning
=	Equal to
!= or < >	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less then
<=	Less then or equal to

Example :

```
SELECCT IT_NAME, AMT
FROM ITEM
WHERE AMT >= 1500;
```

<u>Some more comparison operators are as follows</u> :

BETWEEN Operator

It is a 'range test' operator. It is used to test whether the value is lies between the two specified values. It involves three SQL expressions. The first expression defines the value to be tested; the second and third expressions define the low and high ends of the range to be checked.

Ex. :
```
SELECT * FROM ITEM
WHERE AMT BETWEEN 1000 AND 3000;
```

NOT BETWEEN Operator

This operator is negated version of the BETWEEN operator. This operator is used to test for values that fall outside the specified range.

Ex. :
```
SELECT * FROM  ITEM
WHERE AMT NOT BETWEEN 1000 AND 3000;
```

IN Operator

It is the 'set membership test' operator. It is used to compare single value with a set of values. It tests whether a value of the field is the member of set values. The test expression in an IN test can be any SQL expression, but it's usually just a column name. All of the items in the list of target values must have the same data type, and that type must be comparable to the data type of the test expression.

Ex. :
```
SELECT * FROM ITEM
WHERE IT_NAME IN('PEN', 'PENCIL');
```

NOT IN Operator

It is just opposite of IN operator. You can check if the data value does *not* match any of the target values by using the NOT IN operator. It is used to whether the value of field is not the member of set values.

Ex :
```
SELECT * FROM ITEM
WHERE IT_NAME NOT IN('PEN', 'PENCIL');
```

ANY Operator

It is used to compare a value to each value in a set and returns true if any value is compared according to condition. Comparison can be any of >, <, >=, <=, =, or !=

Ex. :
```
SELECT * FROM ITEM
WHERE AMT > ANY( 200, 400, 500 );
```

List records whose AMT grater than either 200, or 400, or 500

ALL Operator

It is used to compare a value to every value in set and returns true if condition is satisfied for every value.

Ex. :
```
SELECT * FROM ITEM
WHERE AMT > ALL( 200,  400, 500);
```

List records whose AMT greater than 200 and 400 and 500

LIKE operator

This operator is used in pattern matching. This is achieved by using wildcard character (%)

Ex. :
```
SELECT * FROM ITEM
WHERE IT_NAME LIKE 'P%';
```

List records whose IT_NAME starts with character 'P'

NOT LIKE Operator

It is just opposite to LIKE operator.

Ex. :
```
SELECT * FORM ITEM
WHERE IT_NAME NOT LIKE 'P%';
```

List records whose IT_NAME not start with character 'P'

IS NULL Operator

Is NULL operator is used to test whether the value in the column is null.

Ex. :
```
SELECT IT_NAME FROM ITEM
WHERE QTY IS NULL;
```

List IT_NAMES whose QTY is not specified.

IS NOT NULL Operator

It is just reverse of IS NULL operator. It checks not null values.

Ex.:
```
SELECT IT_NAME FROM ITEM
WHERE QTY IS NOT NULL;
```

List IT_NAME whose QTY field is not empty.

EXISTS Operator

EXISTS is an operator that produces a true or false value. It takes sub query as an argument and tests whether a sub query returns any output. It is true if it returns any output (at least one row.) otherwise it is false.

Ex. :
```
 SELECT  * FROM ITEM
WHERE EXISTS (SELECT QTY FROM ITEM WHERE QTY <10);
```

In this example, if any QTY found less than 10 then it will display the contents of ITEM table, otherwise it will display nothing.

NOT EXISTS Operator

It is just reverse of EXISTS operator. It also takes sub query as an argument and tests whether a sub query dose not returns any output. It is true if it does not return any output (not even one row) otherwise it is false.

Ex. :
```
SELECT  * FROM ITEM
WHERE NOT EXISTS(SELECT QTY FROM ITEM WHERE QTY <10);
```

In this example, if any QTY not found less than 10 then it will display the contents of ITEM table, otherwise it will display nothing.

3) Boolean or logical Operators:

Logical operators are used to combine two or more relational expressions or conditions. Following are the logical operators.

AND operator

It combines logical condition. It returns TRUE value if both conditions are true otherwise false.

Ex. :

```
SELECT * FROM  ITEM
WHERE QTY > 20 AND QTY < 100;
```

List record whose QTY is greater than 20 and less than 100.

OR operator

It combines logical condition. It returns true if any of them is true otherwise returns false.

Ex. :

```
SELECT * FROM ITEM
WHERE QTY > 30 OR AMT > 2000;
```

List records whose either QTY > 30 or AMT > 2000 ;

NOT operator

It is used to reverse the result of logical expression.

Ex. : -

```
SELECT  * FROM ITEM
WHERE  NOT (IT-NAME = 'PEN');
```

List records whose IT-NAME is not a Pen.

DDL Commands

The DDL commands are used to create, alter or delete database objects like table and views. Some DDL commands are : `CREATE`, `ALTER`, `DROP` etc.

CREATE TABLE command

Tables are defined with the `CREATE TABLE` command. This command creates an empty table i.e. a table with no rows. Values are entered with the DML command `INSERT`. The `CREATE TABLE` command basically defines a table name as describing a set of named columns in a specified order. It is also defines the data types and sizes of the columns. Each table must have at least one column. The order of columns in the table is determined by the order in which they are specified. The syntax of command is:

```
CREATE TABLE tablename
( columnname1 datatype(size),
  columnname2 datatype(size),
  . . .     );
```

Where, 'tablename' is the name given to new table. 'columename1', 'columnname2' are the names of columns in new table.

Example:

1. Create a student table that contains following columns:

   name with 20 characters, cps with 3 digits, mth with 3 digits,
   etc with 3 digits, total with 3 digits.

```
CREATE TABLE STUDENT
( NAME VARCHAR2(20),
  CPS  NUMBER(3),
 MTH  NUMBER(3),
 ETC  NUMBER(3),
 TOTAL NUMBER(3));
```

• ***Creating a table using a sub-query***

You can create a table by using a SELECT query. The query will create a new table and populate it with the rows selected from the other table. The syntax is :

```
CREATE TABLE tablename
AS
SELECT query;
```

This syntax allows you to create a new table with the same data types as those of the fields that are selected from the existing table. It also allows you to rename the fields in the new table by giving them new names.

When a new table is created with a query, the primary key constraint is not transferred to the new table from the existing table. The NOT NULL constraint does get transferred to the new table.

Example :

```
CREATE TABLE DEPT20
AS
SELECT * FROM DEPT WHERE DEPTNO = 20;
```

In this example, the new table DEPT20 is created. It is based on DEPT table.

ALTER TABLE command

After creating a table, sometimes there is a need to make some changes in its structure. This can be achieved by `ALTER TABLE` command. This command is used to change (alter) the structure of an existing table by adding new columns, modifying existing columns, removing unwanted columns, changing name of column and changing table name.

1) Adding new columns

To add new columns into an existing table the `ALTER TABLE` command used with `ADD` clause and its syntax looks like:

```
ALTER TABLE tablename
ADD (columnname datatype(size),
     columnname datatype(size),
     . . .);
```

Where, 'tablename' is the name of an existing table in which new columns will be added. The column specifications includes the names of new columns their datatypes and sizes.

Examples :

1. To add `percentage` column to the `student` table, following command is used

```
ALTER TABLE STUDENT
ADD (PERCENT NUMBER(5,2));
```

2. Add `result` with `char(5)` and `division` with `char(6)` columns to student table.

```
ALTER TABLE STUDENT
ADD ( RESULT CHAR(5)
      DIVISION CHAR(6));
```

2) Modifying existing columns

To change the datatype and/or size of existing columns of a table, the `ALTER TABLE` command used with `MODIFY` clause.

With an empty column, you can change its datatype and/or size in any way *i.e.* increase or decrease the size. But if the column contains data then you cannot change its datatype and cannot decrease its width. You can only increase the width of that column. The syntax of `ALTER TABLE` command with `MODIFY` clause looks like:

```
ALTER TABLE tablename
MODIFY( columnname datatype(size),
        columnname datatype(size),
        . . .);
```

Examples:

1. Change the width of `name` column of the `student` table to 40 characters.

```
ALTER TABLE STUDENT
MODIFY(NAME VARCHAR2(40));
```

2. Change the datatype of `division` column of `student` table to `varchar2` and change width of `result` column to 10.

```
ALTER TABLE STUDENT
MODIFY( DIVISION VARCHAR2(5), RESULT CHAR(10));
```

3) Dropping a Column

A column can be dropped in an existing table using the `ALTER TABLE` command. One or multiple columns can be dropped at a time. The column may or may not contain any data. When a column or columns is dropped, there must be at least one column left in the table. The column(s) is physically dropped, so it is not possible to recover a dropped column and its data.

a) To drop a single column

The syntax is:

```
ALTER TABLE tablename
DROP COLUMN columnname;
```

Example :- Following example drops the `math` column from the `student` table.

```
ALTER TABLE student
DROP COLUMN math;
```

b) To drop multiple columns

The syntax is:

```
ALTER TABLE tablename
DROP (columnname1, columnname2, ...);
```

Example :- Following example drops the `math`, `etc` and `cps` columns from the `student` table.

```
ALTER TABLE student
DROP (math, etc, cps);
```

4) Rename a Column in table

The `RENAME` keyword can be used with the `ALTER TABLE` statement to rename a column of a table. This is started in Oracle 9i Release 2. The syntax for this is:

```
ALTER TABLE table_name
RENAME COLUMN old_name TO new_name;
```

Where, the '`old_name`' is the name of an existing column which we want to change.

Example: Following renames the `name` column to `stud_name` in the `student` table.

```
ALTER TABLE student
RENAME COLUMN name TO stud_name;
```

5) Rename a table

To change name of any existing table, the Oracle `ALTER TABLE` syntax is:

```
ALTER TABLE table_name
RENAME TO new_table_name;
```

Example: Following renames the table name from `student` to `stud`.

```
ALTER TABLE student
RENAME TO stud;
```

6) To set column as unused

The `DROP` clause with `ALTER TABLE` statement physically deletes the columns. There's another way to delete a column called logical delete, which sets the column as unused and can be deleted later. The syntax is:

```
ALTER TABLE table_name
SET UNUSED COLUMN column_name;
```

After this command, the column will not be accessible anymore. So it won't appear on data views or query results, but the column is still stored in the physical drive. To completely remove the unused set column, run this command:

```
ALTER TABLE table_name
DROP UNUSED COLUMNS;
```

DROP Command

Some times some old tables are no longer needed. Unneeded tables can be completely remove from the memory by using the DROP TABLE command.

DROP TABLE command removes the table definition from the database completely, without a trace. DROP TABLE command deletes whole table i.e. the structure as well as the data from the table. The syntax of DROP TABLE command is:

```
DROP TABLE tablename;
```

Where, the 'table name' is the name of an existing table to be dropped.

When the DROP TABLE statement removes a table from the database, its definition and all of its contents are lost. There is no way to recover the data.

Example :-

```
DROP TABLE STUDENT;
```

RENAME Command

The RENAME command is used to change the name of an existing table. This command sets a new name for any existing table. This command is supported only in Oracle 8*i* version or subsequent versions. A user who is renaming a table must be the owner of that table. The syntax of RENAME command is:

```
RENAME oldtablename TO newtablename;
```

Where, the 'oldtablename' is the name of an existing table which we want to change.

Example :-

```
RENAME std TO STUDENT;
```

DML Commands

The DML commands work directly with data in the tables. Using DML commands one can add, delete, update or retrieve data from the tables. Some DML commands are `INSERT`, `UPDATE`, `SELECT`, `DELETE` etc.

SELECT Command

SELECT Statement (command) is most frequently used SQL command for querying the database and produces a listing of selected records. It can be used for:

1. Displaying the some or all of the columns and rows from a specified one or more tables
2. Displaying calculated information from tables such as average or sums of column values.
3. Combining information from two or more tables.
4. Grouping the data and operating on the groups.
5. Ordering the data
6. Formatting the query results, etc.

The syntax of SELECT statement is:

```
SELECT [ALL|DISTINCT] {*| column1, column2 ...}
FROM table1 [, table2 ...]
[WHERE condition]
[ORDER BY column1 [ASC|DESC] [, column2 [ASC|DESC] ...]
[GROUP BY expression [, expression,  ... ]
[HAVING  condition ] ;
```

Where *column1*, *column2*, . . . use the name of columns of which data is displayed. An asterisk (*) is used as a special qualifier to represent all the columns of table. Table1, table2, . . . give name of tables from which data is retrieved. The `WHERE` clause is optional. When it is included, selected rows will be displayed and when it is omitted; all rows will be displayed. 'condition' in a `WHERE` clause specifies that only those columns or rows be retrieved that fulfils the condition. The `DISTINCT` keyword is used to eliminate duplicate rows of query results.

Examples:-

1. `SELECT * FROM ITEM;`
2. `SELECT IT_NAME,AMT FROM ITEM;`

3.
   ```
   SELECT * FROM ITEM
   WHERE QTY > 50;
   ```
4. `SELECT DISTINCT NAME FROM ITEM;`

5.
   ```
   SELECT SNAME, CITY FROM S, SP
   WHERE S.S# = SP.S#;
   ```

Column Alias

When a `SELECT` query is executed, SQL*Plus uses the column's name as the column heading. Normally the user gives abbreviated names for columns, for example, the attribute name `Lname` is used for an employee's last name.

Column alias are useful because they let you change the column's heading. When a calculated value is displaying, and if a column alias is used, then instead of mathematical expression the column alias is displayed as the column heading. The column alias is written right after the column name with an optional keyword AS in between. The alias heading appears in uppercase by default. Following is the syntax:

```
SELECT columnname [AS] alias . . .
```

If an alias includes spaces or special characters, or you want to preserve its case, you must enclose it in double quotation marks (" ").

Example :

1) In the following example, LastName is an alias for Last and FirstName is an alias for First. Notice that the word AS is omitted the second time, because it is an optional word. The heading will look like

```
SELECT Last AS Lastname, First Firstname FROM student;

                LASTNAME               FIRSTNAME
                ------------           ------------
```

2) If you want the alias uses spaces and special characters, and also want to display it in mixed case, not in uppercase, then you have to enclosed the alias in double quotation marks, as follows :

```
SELECT DeptName AS "Department's Name" FROM dept;

                          Department's Name
                          -------------------
```

INSERT Command

To add data into tables the `INSERT INTO` command is used. When the `INSERT` operation is performed, it creates a new row in the database table; and load the values passed into the columns specified. The syntax is of `INSRT INTO` command is:

```
INSERT INTO tablename[(column list)]
VALUES(constant1, constant2,... );
```

The values in the `INSERT` command are corresponds left to right to the fields defined for the table. That is, the columns of the table and the values have a one to one relationship. Using `INSERT INTO` command you can insert only one record at a time.

You can use `INSERT INTO` command in various ways i.e. to provide values within the command itself or through keyboard. Also you can provide values for all columns of a table for a record or only for selected columns.

- ***To insert values in All columns:***

In this case, no need to indicate the column names in the `INSERT INTO` command. But the values should be given in accordance with the way the columns were created.

Example :- To insert values in ITEM table for all columns the command is:

```
INSERT INTO ITEM
VALUES('PEN', 20,100);
```

- ***To insert values in Particular columns:***

If you want to insert values only in particular columns then you have to indicate these column names and its corresponding values in the command.

Example :- To insert values only in IT_NAME and AMT column of ITEM table.

```
INSERT INTO ITEM(IT_NAME, AMT)
VALUES('LET US C', 900);
```

- ***To insert values through keyboard (from user)***

In this case you do not require to give values for columns into the `INSERT` command. Instead, you have to write name of columns with ampersand (&) symbol at the place of values. So when you execute `INSERT` command, it will ask for values.

Examples :-

1.
   ```
   INSERT INTO ITEM
   VALUES('&IT_NAME', &QTY, &AMT);
   ```

2.
   ```
   INSERT INTO ITEM(IT_NAME, AMT)
   VALUES('&IT_NAME', &AMT);
   ```

• ***Multiple Record Insertion (Using Sub-query)***

Generally the `INSERT` command can insert only one row of the data into the specified table. If required, it can also be used to insert multiple records from another exiting table having equivalent definition. The matching records can be inserted using a select query. The example is:

```
INSERT INTO ITEM_LIST(IT_NAME)
SELECT TI_NAME FROM ITEM
WHERE AMT<1000;
```

DELETE command

The `DELETE` command is used to remove one or more rows from a table. ROLLBACK is possible for `DELETE` command. The syntax is :

```
DELETE FROM tablename
[WHERE condition];
```

A ‘`WHERE`’ clause is optional. If you omit the `WHERE` clause then all the rows will be deleted. If you include `WHERE` clause, then only those rows will be deleted that satisfy the ‘condition’ in `WHERE` clause.

Examples:

1.
   ```
   DELETE FROM ITEM;
   ```

2.
   ```
   DELETE FROM ITEM
   WHERE AMT > 300;
   ```

UPDATE Command

This command is used to change individual columns of all rows or selected rows. Also you can calculate column values using `UPDATE` command. The syntax is:

```
UPDATE tablename
SET column1=Expression [, column2=Expression, ...]
[WHERE condition];
```

Where, ‘Expression’ can be any combination of characters, formulas, or functions that result in the same data type as the specified column. The `WHERE` clause is optional. If `WHERE` clause is included, it specifies a condition that must be met for a row to be updated, If no `WHERE` clause is included, all rows are updated. The `SET` clause determines which columns are updated and what values are stored in them. Only one table can be updated per `UPDATE` command.

Examples:

1.
   ```
   UPDATE STUDENT
   SET TOTAL = PHY + MTH + CPS;
   ```

2.
   ```
   UPDATE ITEM
   SET QTY = 50, AMT = 250
   WHERE TI_NAME = 'PEN';
   ```

ORDER BY Clause

As tables are unordered sets, and, in general the result of SELECT is not in any particular sequence. However, a user has specified that the result is to be ordered in a particular way before being displayed. This can be done be using ORDER BY clause in the SELECT Statement. ORDER BY allows you to impose an order on your output. It orders the query output according to the values in one or more selected columns. The ORDER BY can be used with any number of columns at once. The syntax is:

```
ORDER BY columnname|columnno [ASC | DESC]
[,columnname|columnno [ASC | DESC], . . . ]
```

Where, 'columnname' is the name of the column, and note that, it is always among the columns selected (i.e. columns in the list of SELECT clause). 'ASC' stands for ascending and 'DESC' for descending. It neither ASC nor DESC is specified, and then ASC is assumed by default.

Examples:-

1.
```
SELECT * FROM STUDENT
ORDER BY PER DESC;
```

This displays records from student table in the descending order of percentage.

2.
```
SECECT ROLL_NO, NAME, PER
FROM STUDENT
ORDER BY PER DESC, ROLL_NO;
```

This lists the records from STUDENT table in the descending order of PER and within PER in ascending order of ROLL_NO.

In place of column names, you can use numbers to indicate the columns being used to order the output. These numbers will refer, not to the order of the columns in the table, but to their order in the SELECT list.

Example:-

3.
```
SELECT NAME, PER
FROM STUDENT
ORDER BY 2 DESC;
```

It displays records in descending order of PER.

GROUP BY Clause

By using GROUP BY clause with SELECT Statement we can combine columns and group functions in a single SELECT Statement. The GROUP BY rearranges the table into partitions or groups, such that within any one group all rows have the same value for the GROUP BY field. Then the group functions are applied to each group independently, which produces single-row result for each group. The syntax for SELECT statement using GROUP BY clause is :

```
SELECT groupid1, groupid2, group function (columns)
FROM table(s)
GROUP By groupid1, groupid2;
```

In the SELECT list, a column or combination of columns, constant, or a group function (AVG, SUM, MAX, MIN, COUNT) are specified. Columns which are specified in the SELECT list must be included in the GROUP BY clause. However, a column which is not a part of SELECT clause may also be included in the GROUP BY clause.

Upon execution, firstly the rows of table are grouped into different groups by the GROUP BY column. Then the SELECT clause is applied to each group of the table rather than to each row of the table.

Examples:-

1.
```
SELECT P#, SUM(QTY) FROM SP
GROUP BY P#;
```

It lists the part number and total number of quantities supplied for that part.

2.
```
SELECT P#, S#, SUM(QTY) FROM SP
GROUP BY P#, S#;
```

`HAVING` Clause

`HAVING` is a conditional option that is directly related to the `GROUP BY` option. That means, if `HAVING` is specified; `GROUP BY` should also have been specified. `HAVING` is similar to `WHERE`, in `WHERE` condition we can not use group functions because it evaluates to single row; but `HAVING` does a selection based on the result of group functions & that's why we can use group functions in `HAVING`.

`HAVING` can take only arguments that have a single value per output group. It will always state a condition based on one of the group functions listed in the `SELECT` clause. Columns chosen by `GROUP BY` are also permissible in `HAVING`. `HAVING` uses a result of group function to evaluate whether to omit that group rows from output.

Example:

1.
```
SELECT P#, SUM(QTY) FROM SP
GROUP BY P#
HAVING COUNT(*) > 1;
```

2.
```
SELECT P#, MAX(QTY) FROM SP
WHERE QTY > 200
GROUP BY P#
HAVING SUM(QTY) > 300
ORDER BY 2, P# DESC;
```

JOINS

When the required data exist in more than one table, related tables are joined and data is retrieved like it is taken from single table. Rows of two table are combined based on the given column(s) values. Join is performed using a column from each table as a **connecting column** or **join column**. Connecting columns should have values that match or compare easily, representing the same or similar data in each of the tables participating in the join.

Usually join is performed by using SELECT query. A join condition is specified in the WHERE clause that relates the tables specified in the FROM clause. This condition is referred to as the ***join condition***.

You can select any number of columns from each table. Whenever there is an ambiguity in the column names, you must use qualified names (i.e. **`tablename.columnname`**) to qualify any ambiguous column names.

Types of joins

There are four types of joins:

1) Equi-Join *or* Simple Join
2) Non-Equi Join
3) Outer Join
4) Self Join

1) Equi-Join OR Simple Join

An equi-join is a specific type of join, that uses only equality comparisons (equals to (=) operator) in the join condition.

The equi-join compares each row of one table with each row of another table to find all pairs of rows which satisfy the join condition. When the join condition is satisfied, column values for each matched pair of rows of both tables are combined into a result row. The equi-join returns only those rows from both the tables that have matching values in common columns. The syntax for the equi-join is as follows:

```
SELECT column(s)
FROM table1, table2
WHERE table1.column = table2.column
```

Where,

`Column(s)` : represents the columns from both tables.

`Table1, table2` : represents the tables from which data is retrieved.

`table1.column = table2.column` : represents the join condition.

Example:- Following are two tables named EMPLOYEE and DEPARTMENT.

EMPLOYEE

ENAME	DEPT_ID
Adams	23
Backer	101
Prince	90
White	90
Smith	955

DEPARTMENT

DEPT_ID	DESCI
23	Finance
101	Systems
90	Manufacturing
951	Distribution
1001	Personnel
1209	Marketing

Que. :- Write a query to display employee names along with their department numbers and department names.

Ans. :- Following creates equi join for the EMPLOYEE and DEPARTMENT tables:

```
SELECT EMPLOYEE.ENAME, EMPLOYEE.DEPT_ID, DEPARTMENT.DESCI
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DEPT_ID = DEPARTMENT.DEPT_ID;
```

The output of the above join is:

```
ENAME      DEPT_ID        DESCI
--------   ----------     ---------------
Backer     101            Systems
Adams      23             Finance
Prince     90             Manufacturing
White      90             Manufacturing
```

2) Non-Equi Join

In **non-equi join** the join condition is not equal, which means that the join condition uses operators such as less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=) or any other relational operator.

```
SELECT ENAME, EMPLOYEE.DEPT_ID, DEPARTMENT.DEPT_ID, DESCI
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DEPT_ID > DEPARTMENT.DEPT_ID;
```

```
ENAME      DEPT_ID    DEPT_ID     DESCI
---------  ---------  ---------   -------------
Backer     101         23         Finance
Prince      90         23         Finance
White       90         23         Finance
Smith      955         23         Finance
Backer     101         90         Manufacturing
Smith      955         90         Manufacturing
Smith      955        101         Systems
Smith      955        951         Distribution
```

3) Outer Join

An outer join not only returns the rows from both the tables that have matching values in common columns but also returns all unmatched rows (i.e. those rows from one table that have no match in another table). Nulls are generated for the columns of the table whenever the table has no rows to join to a row in the other table. The syntax for the outer join is as follows:

```
SELECT column(s)
FROM table1, table2
WHERE table1.column = table2.column(+)
```

Where, **(+)** is an outer join operator. The plus sign (+) can be used on one side of the join condition only and not on both sides.

There are two types of outer joins:

i) Left outer join

If the (+) operator is used in the left side of the join condition it is called a left outer join. A left outer join contains all records of the right table, even if the join-condition does not find any matching record in the left table.

ii) Right outer join

If the (+) operator is used in the right side of the join condition it is called a right outer join. A right outer join contains all records of the left table, even if the join-condition does not find any matching record in the right table.

Examples:

Following creates an outer join for the EMPLOYEE and DEPARTMENT tables:

1]

```
SELECT EMPLOYEE.ENAME, DEPARTMENT.DESCI
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DEPT_ID(+) = DEPARTMENT.DEPT_ID;
```

The sample output of the join is :

```
ENAME           DESCI
------          --------------
                Personnel
Backer          Systems
                Marketing
Adams           Finance
Prince          Manufacturing
White           Manufacturing
                Distribution
```

2]

```
SELECT EMPLOYEE.ENAME, EMPLOYEE.DEPT_ID, DEPARTMENT.DESCI
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.DEPT_ID = DEPARTMENT.DEPT_ID(+);
```

The sample output of the join is :

```
ENAME           DESCI
--------        ---------------
Backer          Systems
Adams           Finance
Smith
Prince          Manufacturing
White           Manufacturing
```

3) Self Join

In some situations, you may find it necessary to join a table to itself, as though you were 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 it-self, 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 same, the second table will overwrite the first table and in effect result in only one table being in memory. To avoid this each table is opened under an alias. Now these tables' aliases will cause two identical tables to be opened in different memory locations.

```
SELECT column(s)
FROM table alias1, table alias2
WHERE alias1.column = alias2.column
```

Example :- Retrieve the names of the employees with the names of their respective managers from the EMPLOYEE table.

EMPLOYEE

Employee_No	Name	Manager_No
E00001	Basu Navindgi	E00002
E00002	Rukmini	E00005
E00003	Carol D'Souza	E00004
E00004	Cynthia Bayross	
E00005	Ivan Bayross	

To self join the EMPLOYEE table, the SELECT statement is

```
SELECT EMP.NAME, MNGR.NAME
FROM EMPLOYEE EMP, EMPLOYEE MNGR
WHERE EMP.MANAGER_NO = MNGR.EMPLOYEE_NO;
```

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

The sample output is:

```
EMP.NAME            MNGR.NAME
-------------       ----------
Basu Navindgi       Rukmini
Rukmini             Ivan Bayross
Carol D'Souza       Cynthia Bayross
```

UNION

The UNION operator allows you to combine the output of two or more individual queries. The UNION clause merges the output of two or more queries into a single set of rows and columns. The UNION operator returns all distinct rows retrieved by two queries i.e. it eliminates duplicates while merging rows retrieved by either of the queries. The final output of the UNION clause will be:

```
Output of UNION   =    Records only in query one  +
                       Records only in query two  +
                       A single set of records which
                       is common in both queries
```

There are several restrictions on the tables that can be combined by a UNION operation:

- The two queries must contain the same number of columns.
- The columns selected by the individual SELECT statements must be compatible. i.e. each query must select the same number of columns and each corresponding column must have the same data type.
- Neither of the two tables can be sorted with the ORDER BY clause.

Syntax :

```
Query 1
union
Query 2;
```

Example :- Retrieve the names of the clients and the salesman in the city of Mumbai from the tables CLIENT and SALESMAN.

CLIENT

CN_NO	CNAME	CITY
C001	Ashok Mehra	Mumbai
C002	Prem Iyar	Pune
C003	Rohit Roy	Mumbai

SALESMAN

SM_NO	SNAME	CITY
S001	Manish Patel	Delhi
S002	Kiran Dixit	Mumbai
S003	Mahesh Patil	Pune
S004	Ashok Mehra	Mumbai

Following is the query using UNION to retrieve the names of the clients and the salesman in the city of Mumbai from the tables CLIENT and SALESMAN.

```
SELECT CNAME FROM CLIENT
WHERE CITY = 'Mumbai'
UNION
SELECT SNAME FROM SALESMAN
WHERE CITY = 'Mumbai';
```

This UNION query produces the following result

```
Ashok Mehra
Rohit Roy
Kiran Dixit
```

INTERSECTION

The INTERSECT clause merges the output of two or more queries into a single set of rows and columns. The INTERSECT operator only returns common rows retrieved by both queries. The final output of the INTERSECT clause will be:

```
Output of INTERSECT   =     A single set of records which
                            is common in both queries.
```

There are several restrictions on tables that can be combined by a INTERSECT:

- The two queries must contain the same number of columns.
- The columns selected by the individual SELECT statements must be compatible. i.e. each query must select the same number of columns and each corresponding column must have the same data type.
- Neither of the two tables can be sorted with the ORDER BY clause.

Syntax:

```
Query 1
INTERSECT
Query 2;
```

Example :- Retrieve the names of only those persons who are clients as well as salesmans both and lives in the city Mumbai from the tables CLIENT and SALESMAN.

```
SELECT CNAME FROM CLIENT
WHERE CITY = 'Mumbai'
INTERSECT
SELECT SNAME FROM SALESMAN
WHERE CITY = 'Mumbai';
```

This INTERSECT query produces the following result

```
Ashok Mehra
```

Functions

Functions are special commands that perform computations or processes on a column, constant, value, or group of values. Functions can be used anywhere as an expression or condition. Functions accept user-supplied values called as ***arguments*** and operates on them. SQL functions can be classified as follows:

1) Group Functions (or Aggregate Functions)

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

2) Scalar Functions (or Single Row Functions)

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

Single row functions can be further grouped together by the data type of their arguments and return values. Scalar functions can be classified as:

String Functions	:	Work for *String* data type
Numeric Functions	:	Work for *Number* data type
Conversion Functions	:	Work for *Conversion* of one data type to another
Date Functions	:	Work for *Date* data type

Group Functions *(Aggregate Functions)*

All the previous functions have been based on a single row of data. Oracle also provides functions that operate on groups of rows in a single query. These functions are called "**Group Functions**". Group functions work on a group of rows collected from a table and generally return a numeric value. A group function returns a single result row for a group of queried rows. The general syntax for the group function is:

```
function([ DISTINCT | ALL ] expression)
```

Group functions can operate on either all values of the expression or just the distinct set of values. For each group function, you can specify either `DISTINCT` or `ALL`. If you specify `DISTINCT`, no repeated values will be considered in the calculation. If you specify `ALL` or do not specify anything, each value will be included in the calculation. Following are some group functions:

AVG (expression)

AVG function retrieves the average value for the row returned by the query.

Syntax:

```
AVG([DISTINCT | ALL] expression)
```

Example:1) `SELECT AVG(AMT) FROM ITEM;`

Above command retrieves average value for the AMT column of the table ITEM.

MAX (expression)

MAX function retrieves the highest value from set of rows returned by the query.

Syntax:

```
MAX([ DISTINCT | ALL ] expression)
```

Example: `SELECT MAX(TOTAL) FROM STD;`

Above command returns the highest TOTAL column value from the STD table.

MIN (expression)

MIN function retrieves the lowest value from the set of rows returned by the query.

Syntax:

```
MIN([DISTINCT | ALL] expression)
```

Example: `SELECT MIN(TOTAL) FROM STD;`

Above command returns the lowest TOTAL column value from the STD table.

SUM (expression)

SUM function retrieves the total of all non-null values returned by the query. If all the values in the expression are null, then a null value will be returned.

Syntax:

```
SUM([DISTINCT | ALL] expression)
```

Example: `SELECT SUM(AMT) FROM ITEM;`

Above command returns the total of all row values in the AMT column of the ITEM table.

COUNT (expression)

COUNT function counts the number of entries based on the “expression”. You can specify a column in the 'expression' to retrieve the number of rows of that column in the table that have non-null values.

Syntax:

```
COUNT([ DISTINCT | ALL ] expression)
```

Example: `SELECT COUNT(QTY) FROM ITEM;`

Above counts the number of rows of QTY column that have non-null values.

*You can also use * as 'expression' in the COUNT function. It returns the number of rows in the table, including duplicates and those with nulls.*

```
SELECT COUNT( * ) FROM ITEM;
```

Above command counts the total number of rows in the ITEM table.

Numeric Functions:

Numeric Functions receive numeric inputs and returns a numeric value. Oracle's numeric functions perform mathematical computations on stored database values. These functions represent standard trigonometric and algebraic functions. Following are some numeric functions:

ABS (number)

ABS function returns the absolute value of a numeric value. The absolute value is the numeric portion of the number without a sign.

Syntax :

```
ABS(number)
```

Where, 'number' is any numeric value, expression or column name.

Ex.:

1) `SELECT ABS(-25) FROM DUAL;`

Above command returns 25 i.e. absolute value of -25

2) `SELECT ABS(AMT) FROM ITEM;`

Above command return absolute values of all the rows of AMT column.

MOD (n, m)

MOD function returns the modulus of a number n, with respect to the divisor m. The result of this function is the remainder of the integer division of n/m.

Syntax :

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

Where, 'n' is the number and 'm' is the divisor.

Ex.:

1) `SELECT MOD(17,4) FROM DUAL;`

Above command returns 1 i.e. remainder of 17/4.

FLOOR (number)

FLOOR function returns the largest integer value that is less than or equal to the input value. That is, the FLOOR function *'round down'* the given numeric value. If the input number is negative, the result will be a number with a higher negative magnitude.

Syntax :

```
FLOOR(number)
```

Where, ‘number’ is any numeric value, expression or column name.

Ex.:

1) `SELECT FLOOR(5.4) FROM DUAL;`

Above command return 5.

2) `SELECT FLOOR(AMT) FROM ITEM;`

Above command will round down all rows of AMT column.

CEIL (number)

CEIL function returns the next integer value greater than or equal to the input value. That is, the CEIL function '***round up***' the given numeric value to its next integer.

Syntax :

```
CEIL(number)
```

Where, 'number' is any numeric value, expression or column name.

Ex.:

1) `SELECT CEIL(5.4) FROM DUAL;`

Above command return 6, the next integer of 5.4.

2) `SELECT CEIL(AMT) FROM ITEM;`

Above command will round up all rows of AMT column to next integer value.

TRUNC (number [, decimal_places])

TRUNC function truncates a numerical value at the specified number of decimal places. If 'decimal_places' is not specified, TRUNC assumes zero decimal-places and the number is truncated to an integer. If the decimal_places value is negative, the input value will be truncated at the defined power of 10.

Syntax:

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

Ex. :

```
1)  SELECT TRUNC(41.7374,2) FROM DUAL;
```

Above command returns 41.73.

```
2)  SELECT TRUNC(41.7374,0) FROM DUAL;
```

Above command returns 41.

SQRT (number)

SQRT function returns the Square Root of the input value, which must be 0 (zero) or a positive number. If a negative number is given, SQRT returns an error.

Syntax :

```
SQRT(number)
```

Where, 'number' should not be negative.

Ex.:

1) `SELECT SQRT(144) FROM DUAL;`

Above command returns 12 i.e. square root of 144.

SIGN (number)

SIGN function returns value of 1 if the input value is a positive number and -1 if the input value is negative. If the input value is 0 (zero), SIGN returns 0 (zero). You can use this function to compare two numbers.

Syntax:

```
SIGN(number)
```

Where, 'number' is any numeric value, expression or column name.

Ex.:

```
1)  SELECT SIGN(20) FROM DUAL;
```

Above command return 1.

```
2)  SELECT SIGN(40-65) FROM DUAL;
Above command returns -1.
```

POWER (X, Y)

POWER function returns X raised to the Y^{th} power, i.e. X^Y .

Syntax :

```
POWER(X, Y)
```

Ex.:

```
1)  SELECT POWER(2, 5) FROM DUAL;
```

Above command returns value 32. i.e. 2^5.

ROUND (number, decimal_digits)

ROUND function rounds a numerical value to the nearest number of decimal digits. If 'decimal_ digits' specified is zero, the returned value is an integer. If the decimal_digits value is negative, the input value will be rounded at the nearest power of 10.

Syntax:

```
ROUND(number, decimal_digits)
```

Ex.:

```
1)  SELECT ROUND(587.653,1) FROM DUAL;
```

Above command return 587.7.

```
2)  SELECT ROUND(587.653, -1) FROM DUAL;
```

Above command return 590.

Character Functions

The Oracle Character functions operate on character string input values and return either another character string or a numeric value depending on the respective function. These functions are typically used to retrieve information about the string or alter the way the string displayed. Some character functions are:

INITCAP (string)

INITCAP function returns the input string with the first letter of each word in uppercase. All other characters appear in lowercase..

Syntax :

```
INITCAP(string)
```

Ex.:

```
1)  SELECT INITCAP('welcome to oracle' ) FROM DUAL;
```

Above command returns Welcome To Oracle.

LOWER (string)

LOWER function returns the input string in all lowercase letters. For numbers and other non-alphanumeric characters, no conversion is performed

Syntax:

```
LOWER(string)
```

Ex.:

```
1)  SELECT LOWER('COMputeR SCIENCE') FROM DUAL;
```

Above command returns computer science.

```
2)  SELECT LOWER(NAME)FROM STD;
```

Above command returns all the names in lowercase.

UPPER (string)

UPPER function returns the input string in all uppercase letters. For numbers and other non-alphanumeric characters, no conversion is performed.

Syntax :

```
UPPER(string)
```

Ex.:

```
1)  SELECT UPPER('COMputeR SCIENCE') FROM DUAL;
```

Above command returns COMPUTER SCIENCE.

```
2)  SELECT UPPER(NAME) FROM STD;
```

Above command returns all the names in uppercase.

INSTR (string, search_string [, n [, m]])

INSTR function returns the character position number of m^th occurrence of a search_string embedded within the input string starting at position 'n'. If the search_string is not found, INSTR returns a value 0 (zero).

Syntax:

```
INSTR(string, search_string [, n [, m ] ])
```

Ex.:

```
1) SELECT INSTR('COORPORATION FLOOR','OR',1,2) FROM DUAL;
```

Above command will search the string 'OR' in 'COORPORATION FLOOR' from the first character and return the character position number when the second time 'OR' occurs. So the output of above command is 6.

```
2) SELECT INSTR('COORPORATION FLOOR','OR',8,2 )FROM DUAL;
```

Above command return 0.

SUBSTR (string, start [, length])

SUBSTR function returns a sub string from the input string , starting at a character number 'start' for the specified 'length' of characters. If no 'length' is specified, SUBSTR defaults to the end of the input string.

Syntax :

```
SUBSTR(string, start [, length])
```

Ex.:

```
    1)  SELECT SUBSTR( 'COMPUTER', 4, 3) FROM DUAL;
```

Above command returns the string 'PUT'. In this command, 4 is a starting position of sub string and 3 is a number of characters to be read from starting position.

```
    2)  SELECT SUBSTR( 'COMPUTER', 4 ) FROM DUAL;
```

Above command returns PUTER.

LENGTH (string)

LENGTH function returns the number of characters in the input string.

Syntax:

```
LENGTH(string)
```

Ex.:

```
    1)  SELECT LENGTH( 'HELLO') FROM DUAL;
```

Above command return 5 i.e. the number of character in the string HELLO.

LTRIM (string [, set_of_chars])

LTRIM function removes all characters from the left of '*string*' up to the first character not in the '*set_of_chars*'. Every character in the set is compared to the beginning character in the string without consideration of the order of the set. If no character set is specified, LTRIM removes all leading spaces from the input string.

Syntax:

```
LTRIM(string [, set_of_chars])
```

Ex.:

```
     1)  SELECT LTRIM('PROM', 'P' ) FROM DUAL;
```

Above command removes the character 'P' from the left of the string 'PROM' and returns the output ROM.

```
     2)  SELECT LTRIM('   SCIENCE' ) FROM DUAL;
```

Above command returns SCIENCE i.e. it removes all leading spaces of SCIENCE.

```
     3)  SELECT LTRIM ( 'XYZABC' , 'XY' ) FROM DUAL;
```

Above command removes XY from the string 'XYZABC' and returns ZABC.

RTRIM (string [, set_of_chars])

RTRIM function removes all characters from the right side of the input 'string'. It removes all final characters from string after the last character not in the set. Set_of_chars is optional. It defaults to spaces.

Syntax:

```
RTRIM( string [, set_of_chars ] )
```

Ex.:

```
    1)  SELECT RTRIM('PROM', 'M' ) FROM DUAL;
```

Above command removes the character 'M' from the string 'PROM' and returns the output PRO.

```
    2)  SELECT RTRIM( 'XYZABC', 'BC' ) FROM DUAL;
```

Above command removes BC from the siring 'XYZABC' and returns XYZA.

LPAD (string, n [, pad_char])

LPAD function pads the left side of the input string with repetitions of the 'pad_chars' to return a character string that is 'n' characters long.

Syntax:

```
LPAD(string, n [, pad_char ] )
```

Ex.:

```
1) SELECT LPAD('KUMAR', 10, '*' ) FROM DUAL;
```

Above command returns KUMAR with leading asterisks to make the resulting string ten characters long. This command gives output *****KUMAR.

```
2) SELECT LPAD(TO_CHAR( AMT ), 10, '*' ) FROM DUAL;
```

Above command will print the amount with leading asterisks to make the resulting string ten characters long.

RPAD (string, n [, pad_char])

RPAD function pads the right side of the input string with repetitions of the 'pad chars' to return a character string that is 'n' characters long.

Syntax:

```
RPAD(string, n [, pad_char])
```

Ex.:

```
1)  SELECT RPAD('KUMAR', 10, '*' ) FROM DUAL;
```

Above command returns output KUMAR*****.

CONCAT( )

CONCAT function concatenates two strings and returns the concatenated single string. It works like this:

Syntax :

```
CONCAT(string1, string2)
```

Where, 'string1' and 'string2' are character strings or character columns.

Ex. :

```
1) SELECT CONCAT('COMPUTER ', 'SCIENCE') FROM DUAL;
```

Output of above query will be

```
CONCAT('COMPUTER
----------------
COMPUTER SCIENCE
```

CHR( )

CHR returns the character equivalent of the number it uses as an argument. The character it returns depends on the character set (ASCII) of the database.

Syntax :

```
CHR(number)
```

Where, '*number*' is a numeric value or numeric column.

Ex. :

```
    1)  SELECT CHR(65) FROM DUAL;
```

Output of above query will be

```
              CHR(65)
              -------
                 A
```

Date functions

The date functions are used to manipulate on date and time values.

Add_Months()

The `ADD_MONTHS()` function adds the specified number of months to a date value and returns a date with a specified number of months added.

Syntax:

```
ADD_MONTHS(date, number_months)
```

Where,

The `date` argument is a date value or any expression that evaluates to a `DATE` value to which the number of month is added and the `number_months` argument is an integer that represents a number of months which adds to the date argument. Value of this argument may be positive, negative or zero.

Example:

```
select add_months('26-jan-2019', 5) from dual;

ADD_MONTH
---------
26-JUN-19
```

Last_Day()

The `Last_Day()` function takes a `DATE` argument and returns the date of the last day of the corresponding month of that date.

Syntax:

```
LAST_DAY(date)
```

Where, `Date` is any valid date of which you want to get the last day of the month.

Example:

```
select last_day('2-feb-2019') from dual;

LAST_DAY(
---------
28-FEB-19
```

Months_Between()

`The MONTHS_BETWEEN` function returns number of months lies between the two dates, *date1* and *date2*. If *date1* is later than *date2*, then the result is positive.

If *date1* is earlier than *date2*, then the result is negative. If *date1* and *date2* are either the same days of the month or both last days of months, then the result is always an integer. Otherwise number of months are calculated the fractional portion of the result based on a 31-day month.

Syntax:

```
MONTHS_BETWEEN(date1, date2)
```

Where, `Date1` argument is first date and `date2` argument is a second date.

Examples:

1) `select months_between('1-Jan-2019', '31-Dec-2019') from dual;`

```
MONTHS_BETWEEN('1-JAN-2019','31-DEC-2019')
------------------------------------------
                                 -11.96774
```

2) `select months_between('31-Dec-2019', '1-Jan-2019') from dual;`

```
MONTHS_BETWEEN('31-DEC-2019','1-JAN-2019')
------------------------------------------
                                 11.967742
```

SYSDATE

The `SYSDATE` function is used to get the current date and time set for the operating system on which the local database is stored. The `SYSDATE` function requires no arguments.

Syntax:

```
SYSDATE
```

Example:

```
SELECT SYSDATE FROM DUAL;
```

■ ■ ■ ■ ■