SQL [STRUCTURED QUERY LANGUAGE]

SQL is a standard language for accessing and manipulating databases.

► What can SQL do?
SQL can execute queries against a database
SQL can retrieve data from a database
SQL can insert records in a database
SQL can update records in a database
SQL can delete records from a database
SQL can create new databases
SQL can create new tables in a database
SQL can create stored procedures in a database
SQL can create views in a database
SQL can set permissions on tables, procedures, and views

► SQL Commands:
SQL commands are instructions. It is used to communicate with the database. It is also used to perform specific tasks, functions, and queries of data.
SQL can perform various tasks like create a table, add data to tables, drop the table, modify the table, set permission for users.

► Types of SQL Commands :
There are five types of SQL commands: DDL, DML, DCL, TCL, and DQL.

1. DDL - Data Definition Language : DDL changes the structure of the table like creating a table, deleting a table, altering a table, etc.

Commands that come under DDL :
● **CREATE** : It is used to create a new table in the database.
Syntax : `CREATE TABLE TABLE_NAME (COLUMN_NAME DATATYPES[,....]);`
Example : `CREATE TABLE EMPLOYEE(Name VARCHAR2(20), Email VARCHAR2(100), DOB DATE);`

● **DROP** : It is used to delete both the structure and record stored in the table.
Syntax : `DROP TABLE ;`
Example : `DROP TABLE EMPLOYEE;`

● **ALTER** : It is used to alter the structure of the database.
Syntax :
To add a new column in the table :
```
ALTER TABLE table_name ADD column_name COLUMN-definition;
```

To modify the existing column of the table :
```
ALTER TABLE MODIFY(COLUMN DEFINITION....);
```

Example :
```
ALTER TABLE STU_DETAILS ADD(ADDRESS VARCHAR2(20));
```
Or,
```
ALTER TABLE STU_DETAILS MODIFY (NAME VARCHAR2(20));
```

● **TRUNCATE**: It is used to delete all the rows from the table and free the space containing the table.
Syntax : `TRUNCATE TABLE table_name;`
Example : `TRUNCATE TABLE EMPLOYEE;`

2. DML - Data Manipulation Language : DML commands are used to modify the database. It is responsible for all form of changes in the database.

Commands that come under DML :
● **INSERT** : It is used to insert data into the row of a table.
Syntax :
```
INSERT INTO TABLE_NAME
(col1, col2, col3,.... col N)
VALUES (value1, value2, value3, .... valueN);
```

or,

```
INSERT INTO TABLE_NAME
VALUES (value1, value2, value3, .... valueN);
```

Example :
```
INSERT INTO ML_basics-to-advance (Author, Subject) VALUES ("Sonoo",
"DBMS");
```

● **UPDATE** : This command is used to update or modify the value of a column in the table.
Syntax : `UPDATE table_name SET [column_name1= value1,...column_nameN = valueN] [WHERE CONDITION];`

Example :
```
UPDATE students
SET User_Name = 'Sonoo'
WHERE Student_Id = '3';
```

● **DELETE**: It is used to remove one or more row from a table.
Syntax : `DELETE FROM table_name [WHERE condition];`
Example : `DELETE FROM ML_basics-to-advance  WHERE Author="Sonoo";`

3. DCL - Data Control Language : DCL commands are used to grant and take back authority from any database user.

Commands that come under DCL :
● **GRANT** : It is used to give user access privileges to a database.
Syntax : `GRANT SELECT, UPDATE ON MY_TABLE TO SOME_USER, ANOTHER_USER;`

● **REVOKE** : It is used to take back permissions from the user.
Syntax : `REVOKE SELECT, UPDATE ON MY_TABLE FROM USER1, USER2;`

4. TCL - Tansaction Control Language : TCL commands can only use with DML commands like INSERT, DELETE and UPDATE only. These operations are automatically committed in the database that's why they cannot be used while creating tables or dropping them.

Commands that come under TCL :
● **COMMIT** : Commit command is used to save all the transactions to the database.
Syntax : `COMMIT;`
Example :
```
DELETE FROM CUSTOMERS
WHERE AGE = 25;
COMMIT;
```

● **ROLLBACK** : Rollback command is used to undo transactions that have not already been saved to the database.
Syntax : `ROLLBACK;`
Example :
```
DELETE FROM CUSTOMERS  WHERE AGE = 25;
ROLLBACK;
```

● **SAVEPOINT** : It is used to roll the transaction back to a certain point without rolling back the entire transaction.
Syntax : `SAVEPOINT SAVEPOINT_NAME;`

5. DQL - Data Query Language : DQL is used to fetch the data from the database.

Command that comes under DQL :
● **SELECT** : This is the same as the projection operation of relational algebra. It is used to select the attribute based on the condition described by WHERE clause.
Syntax :
```
SELECT expressions
FROM TABLES
WHERE conditions;
```

Example :
```
SELECT emp_name
FROM employee
WHERE age > 20;
```

► **JOINS in SQL :**
The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.

There are different types of joins available in SQL –
1. Inner Join
2. Left Join
3. Right Join
4. Full Join
5. Self Join
6. Cartesian Join

1. Inner Join : The most important and frequently used of the joins is the **INNER JOIN**. They are also referred to as an **EQUIJOIN**. The INNER JOIN creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate.

Syntax :
```
SELECT table1.column1, table2.column2...
FROM table1
INNER JOIN table2
ON table1.common_field = table2.common_field;
```

Example :
Consider two tables are there,
Table 1 – CUSTOMERS table is as follows.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Table 2 – ORDERS Table as follows.

```
+-----+---------------------+-------------+--------+
| OID | DATE                | CUSTOMER_ID | AMOUNT |
+-----+---------------------+-------------+--------+
| 102 | 2009-10-08 00:00:00 |           3 |   3000 |
| 100 | 2009-10-08 00:00:00 |           3 |   1500 |
| 101 | 2009-11-20 00:00:00 |           2 |   1560 |
| 103 | 2008-05-20 00:00:00 |           4 |   2060 |
+-----+---------------------+-------------+--------+
```

Now let's join together using the INNER JOIN clause.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   INNER JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce this result.

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

2. Left Join : returns all rows from the left table, even if there are no matches in the right table.

Syntax :

```
SELECT table1.column1, table2.column2...
FROM table1
LEFT JOIN table2
ON table1.common_field = table2.common_field;
```

Example :
Consider those previous tables and use the Left Join here.

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   LEFT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   NULL | NULL                |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   NULL | NULL                |
|  6 | Komal    |   NULL | NULL                |
|  7 | Muffy    |   NULL | NULL                |
+----+----------+--------+---------------------+
```

3. Right Join : returns all rows from the right table, even if there are no matches in the left table.
Syntax :

```
SELECT table1.column1, table2.column2...
FROM table1
RIGHT JOIN table2
ON table1.common_field = table2.common_field;
```

Example :

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   RIGHT JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

4. Full Join : It combines both the results of the RIGHT JOIN and LEFT JOIN.
Syntax :

```
SELECT table1.column1, table2.column2...
FROM table1
FULL JOIN table2
ON table1.common_field = table2.common_field;
```

Example :

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS
   FULL JOIN ORDERS
   ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

This would produce the following result.

```
+------+----------+--------+---------------------+
| ID   | NAME     | AMOUNT | DATE                |
+------+----------+--------+---------------------+
|    1 | Ramesh   |   NULL | NULL                |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|    5 | Hardik   |   NULL | NULL                |
|    6 | Komal    |   NULL | NULL                |
|    7 | Muffy    |   NULL | NULL                |
|    3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|    3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|    2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|    4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
+------+----------+--------+---------------------+
```

5. Self Join : The SQL **SELF JOIN** is used to join a table to itself as if the table were two tables; temporarily renaming at least one table in the SQL statement.

Syntax :

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

Example :

```
SQL> SELECT  a.ID, b.NAME, a.SALARY
   FROM CUSTOMERS a, CUSTOMERS b
   WHERE a.SALARY < b.SALARY;
```

This would produce the following result and here we have used the CUSTOMER table only.

```
+----+----------+---------+
| ID | NAME     | SALARY  |
+----+----------+---------+
|  2 | Ramesh   | 1500.00 |
|  2 | kaushik  | 1500.00 |
|  1 | Chaitali | 2000.00 |
|  2 | Chaitali | 1500.00 |
|  3 | Chaitali | 2000.00 |
|  6 | Chaitali | 4500.00 |
|  1 | Hardik   | 2000.00 |
|  2 | Hardik   | 1500.00 |
|  3 | Hardik   | 2000.00 |
|  4 | Hardik   | 6500.00 |
|  6 | Hardik   | 4500.00 |
|  1 | Komal    | 2000.00 |
|  2 | Komal    | 1500.00 |
|  3 | Komal    | 2000.00 |
|  1 | Muffy    | 2000.00 |
|  2 | Muffy    | 1500.00 |
|  3 | Muffy    | 2000.00 |
|  4 | Muffy    | 6500.00 |
|  5 | Muffy    | 8500.00 |
|  6 | Muffy    | 4500.00 |
+----+----------+---------+
```

6. Cartesian Join : returns the Cartesian product of the sets of records from the two or more joined tables.

Syntax :

```
SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]
```

Example :

```
SQL> SELECT  ID, NAME, AMOUNT, DATE
   FROM CUSTOMERS, ORDERS;
```

This would produce the following result.

```
+----+----------+--------+---------------------+
| ID | NAME     | AMOUNT | DATE                |
+----+----------+--------+---------------------+
|  1 | Ramesh   |   3000 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1500 | 2009-10-08 00:00:00 |
|  1 | Ramesh   |   1560 | 2009-11-20 00:00:00 |
|  1 | Ramesh   |   2060 | 2008-05-20 00:00:00 |
|  2 | Khilan   |   3000 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1500 | 2009-10-08 00:00:00 |
|  2 | Khilan   |   1560 | 2009-11-20 00:00:00 |
|  2 | Khilan   |   2060 | 2008-05-20 00:00:00 |
|  3 | kaushik  |   3000 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1500 | 2009-10-08 00:00:00 |
|  3 | kaushik  |   1560 | 2009-11-20 00:00:00 |
|  3 | kaushik  |   2060 | 2008-05-20 00:00:00 |
|  4 | Chaitali |   3000 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1500 | 2009-10-08 00:00:00 |
|  4 | Chaitali |   1560 | 2009-11-20 00:00:00 |
|  4 | Chaitali |   2060 | 2008-05-20 00:00:00 |
|  5 | Hardik   |   3000 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1500 | 2009-10-08 00:00:00 |
|  5 | Hardik   |   1560 | 2009-11-20 00:00:00 |
|  5 | Hardik   |   2060 | 2008-05-20 00:00:00 |
|  6 | Komal    |   3000 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1500 | 2009-10-08 00:00:00 |
|  6 | Komal    |   1560 | 2009-11-20 00:00:00 |
|  6 | Komal    |   2060 | 2008-05-20 00:00:00 |
|  7 | Muffy    |   3000 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1500 | 2009-10-08 00:00:00 |
|  7 | Muffy    |   1560 | 2009-11-20 00:00:00 |
|  7 | Muffy    |   2060 | 2008-05-20 00:00:00 |
+----+----------+--------+---------------------+
```

► SQL Constraints :

Constraints are the rules enforced on the data columns of a table. These are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the database.

Many types of constraints are there –

● **NOT NULL CONSTRAINT -** Ensures that a column cannot have NULL value.

Example :

```
CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID));
```

● **DEFAULT CONSTRAINT -** Provides a default value for a column when none is specified.
Example :

```
CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
   AGE  INT              NOT NULL,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2) DEFAULT 5000.00,
   PRIMARY KEY (ID)
);
```

● **UNIQUE CONSTRAINT -** Ensures that all values in a column are different.
Example :

```
CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
   AGE  INT              NOT NULL UNIQUE,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);
```

● **PRIMARY KEY -** Uniquely identifies each row/record in a database table.
Example :

```
CREATE TABLE CUSTOMERS(
   ID   INT              NOT NULL,
   NAME VARCHAR (20)     NOT NULL,
   AGE  INT              NOT NULL,
   ADDRESS  CHAR (25) ,
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);
```

● **FOREIGN KEY -** Uniquely identifies each row/record in any of the given database table.
Example :

```
CREATE TABLE ORDERS (
   ID          INT        NOT NULL,
   DATE        DATETIME,
   CUSTOMER_ID INT references CUSTOMERS(ID),
   AMOUNT     double,
   PRIMARY KEY (ID)
);
```

● **CHECK CONSTRAINT -** The CHECK constraint ensures that all the values in a column satisfies certain conditions.
Example :

```
ALTER TABLE CUSTOMERS
   MODIFY AGE INT NOT NULL CHECK (AGE >= 18 );
```

► ORDER BY CLAUSE :

The SQL **ORDER BY** clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default.

Syntax :

```
SELECT column-list
FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

Example :

Using the previous table named as CUSTOMERS. The query goes as follows.

```
SQL> SELECT * FROM CUSTOMERS ORDER BY NAME, SALARY;
```

This would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
+----+----------+-----+-----------+----------+
```

Now, for descending order, the query is as follows.

```
SQL> SELECT * FROM CUSTOMERS ORDER BY NAME DESC;
```

This would produce the following result.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
+----+----------+-----+-----------+----------+
```

► GROUP BY CLAUSE :

The SQL **GROUP BY** clause is used in collaboration with the SELECT statement to arrange identical data into groups. This GROUP BY clause follows the WHERE clause in a SELECT statement and precedes the ORDER BY clause.

Syntax :

```
SELECT column1, column2
FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

Example :

Using the CUSTOMERS table for the example.

```
SQL> SELECT NAME, SUM(SALARY) FROM CUSTOMERS
   GROUP BY NAME;
```

This code will produce the following result.

```
+----------+-------------+
| NAME     | SUM(SALARY) |
+----------+-------------+
| Chaitali |     6500.00 |
| Hardik   |     8500.00 |
| kaushik  |     2000.00 |
| Khilan   |     1500.00 |
| Komal    |     4500.00 |
| Muffy    |    10000.00 |
| Ramesh   |     2000.00 |
+----------+-------------+
```

-- O --