Velocity Corporate

Training Centre

SQL FOR Data Science

Content:

Velocity Corporate Training Centre Pune.

- **SUBQUERIES**
- **SQL JOINS**
    1. INNER JOIN
    2. LEFT JOIN
    3. RIGHT JOIN
    4. FULL JOIN
- **ANALYTICAL FUNCTIONS**
    1. ROW_NUMBER ()
    2. LEAD ()
    3. LAG ()
    4. RANK
    5. DENSE_RANK ()

DATABASE:

- A database is an organized collection of data, stored in a computer system.
- It is an electronic system that makes data access, manipulation, retrieval, storage, and management very easy.

DBMS:

DBMS is the set of application program used to access, update and manage the data.
The goal of DBMS is to provide an environment that is both convenient and efficient to use for :

- Storing data into the database.
- Retrieving data from the database

RDBMS:

- It is a database management system based on a relational model. It facilitates you to store, access, and manipulate the data stored in relational databases.
- Relational database stores the data in the collection of tables.
- Examples of relational database management systems are MySQL, SQL Server, Oracle, PostgreSQL, etc.

DATABASE TABLES:

Data in relational databases is stored in the form of tables. A database often contains one or multiple tables. Each table is defined with a unique name and contains rows and columns.

Example - Employee table:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	450000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

The table above contains 5 rows/records (one for each employee) and 4 columns (emp_id, emp_name, salary, location).

STRUCTURED QUERY LANGUAGE (SQL):

- SQL is a **non procedural** programming language designed for managing data in relational database management systems (RDBMS).
- SQL was originally developed by IBM for querying, defining and altering relational databases.
- SQL is **not case-sensitive**. Generally, SQL keywords are written in upper case.
- Using SQL, we can query our database in several ways using English like statements, every SQL statement is called **SQL Query**.
- SQL follows some unique set of rules and guidelines called **syntax**.
- every SQL query ends with a semicolon (;). It is a standard way in a database system in which more than one SQL statement is used in the same call.
- All RDBMS like MySQL, Oracle, SQL Server, Sybase, PostgreSQL, and SQL Server use SQL as a standard database language

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 and database objects
- SQL can set permissions on tables and views

- **Using Comments in SQL**

Using comments in the SQL script is important to make the script easier to read and understand.

In SQL we can use 2 different kinds of comments:

- Single-line comment
- Multiple-line comment

1. Single-line comment

We can comment one line at the time using "--" before the text you want to comment out.

Syntax: -- text_of_comment

2. Multiple-line comment

We can comment several line using "/*" in the start of the comment and "*/" in the end of the comment.

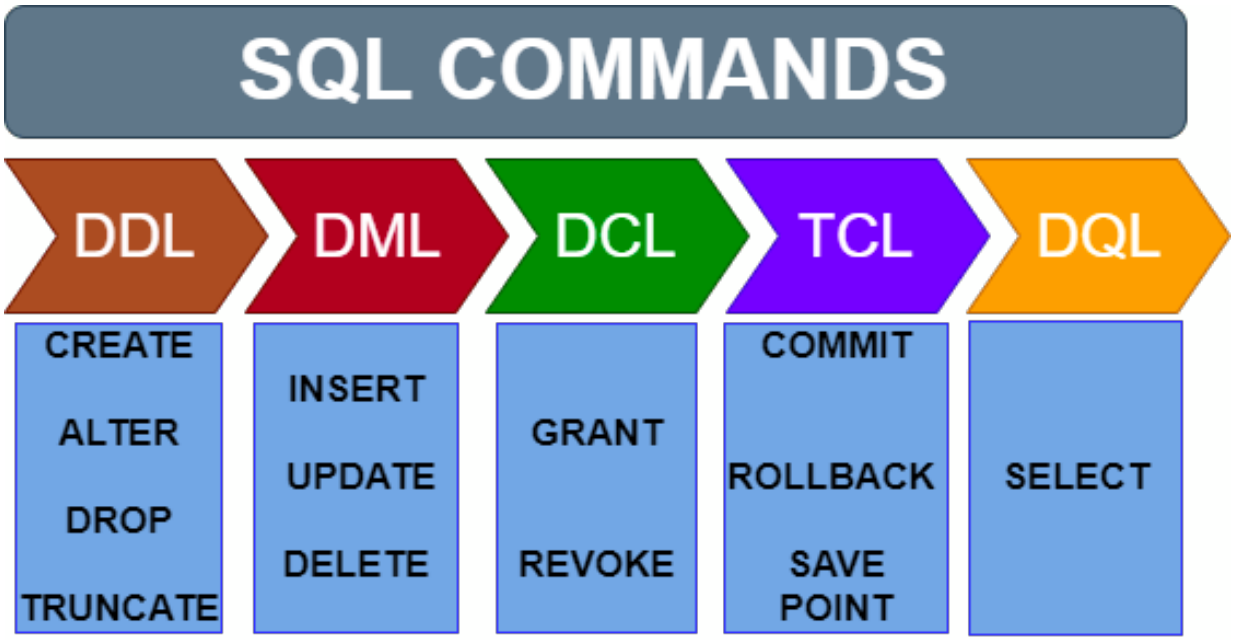
Syntax:

```
/* text_of_comment text_of_comment */
```

SQL COMMANDS:

SQL commands are instructions. these are used to communicate with the database and perform CRUD operations (Create, Read, Update, Delete).

Types of SQL commands:



There are five types of SQL commands as below:

1. DDL (Data Definition Language):

The **Data Definition Language (DDL)** manages the structure.
DDL Commands: **CREATE, ALTER, TRUNCATE ,DROP**

SQL	Description
CREATE	Creates a database and database objects like tables and views
DROP	Deletes an object from a database
TRUNCATE	Deletes the entire data from a table
ALTER	Modifies the structure of an existing object in various ways, ex. Adding a column

2. DML (Data Manipulation Language):

These commands are used to manipulate/modify the data stored in a database.
DML Commands: **INSERT, UPDATE, DELETE**

3. DCL (Data Control Language):

These commands are used to grant or revoke database access. Only database administrators have the access to perform these activities.
DCL Commands : **GRANT, REVOKE**

4. TCL (Transaction Control Language):

These commands are used to manage a database transaction. A transaction is a group of tasks that can have multiple INSERTs, DELETEs, and UPDATEs.
TCL Commands : **COMMIT, ROLLBACK, SAVEPOINT**

5. DQL (Data Query Language):
This command is used to read data from database.
DQL Command : **SELECT**

1. DATA DEFINITION LANGUAGE (DDL):

1.1. CREATE:
Create command is used to create database, and database objects like tables and views.

1.1.1. CREATE DATABASE:
Syntax:

```
CREATE DATABASE Database_name;
```

Example: CREATE DATABASE db_dev;

To start using this database created above, we need to use below syntax:
Syntax:

```
USE database_name;
```

Example:
USE db_dev;

1.1.2. CREATE TABLE:

CREATE TABLE statement is used to create a table with its defined structure.

```
CREATE TABLE Table_name
(
Column1 datatype,
Column2 datatype,
…);
```

The column parameter specifies the name of the columns/fields in the table.
The datatype parameter specifies the type of data a column can hold (e.g. varchar, char, int, date, etc)

Example:
The following example creates an employee table that contains four columns emp_id, emp_name, salary & location.

```
CREATE TABLE employee
(emp_id int,
emp_name varchar(10),
salary int,
location varchar(20));
```

An empty employee table will now look like below:

Table name: employee

Emp_id	emp_name	Salary	Location

To check the structure of any database table, we can use DESC command.
DESC stands for describe. This basically gives the structural details of an existing table

Syntax:

```
DESC Table_name;
```

Example: DESC employee;

Output:

FIELD	TYPE	NULL	KEY	DEFAULT	EXTRA
Emp_id	Int	YES		NULL	
emp_name	varchar(10)	YES		NULL	
Salary	Int	YES		NULL	
Location	varchar(20)	YES		NULL	

Below query will help us get the definition of existing database table

```
SHOW CREATE TABLE Table_name;
```

Example:

SHOW CREATE TABLE employee;

Output:

```
CREATE TABLE `employee` (
  `emp_id` int DEFAULT NULL,
  `emp_name` varchar(10) DEFAULT NULL,
  `salary` int DEFAULT NULL,
  `location` varchar(20) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

1.2. ALTER:

- ALTER statements can be used to modify the existing table structure.
- Modifications, like adding, renaming, and deleting a column or modifying column's datatype can be done using this command.

1.2.1. ADD COLUMN:

Syntax: To ADD COLUMN at the end of the table

```
ALTER TABLE table_name ADD COLUMN column_name datatype;
```

Syntax: To add column as the first column of the table, use below syntax

```
ALTER TABLE table_name ADD COLUMN column_name datatype FIRST;
```

Syntax: To add a column after a specific column in a table, use below syntax :

```
ALTER TABLE table_name ADD COLUMN column_name datatype AFTER existing_column_name;
```

1.2.2. MODIFY COLUMN

Syntax: To change the data type of a column in a table

```
ALTER TABLE table_name MODIFY COLUMN column_name new_datatype;
```

1.2.3. RENAME COLUMN

Syntax: To rename a column ,use below syntax:

```
ALTER TABLE table_name RENAME COLUMN existing_column_name TO new_column_name;
```

1.2.4. DROP COLUMN

Syntax: To remove a column from table, use below syntax:

```
ALTER TABLE table_name DROP COLUMN column_name;
```

1.3. TRUNCATE:

The TRUNCATE statement can be used to DELETE all rows from table/empty the table.

Syntax:

```
TRUNCATE TABLE table_name;
```

1.4. DROP:

A DROP statement is used to delete/drop a database or other database objects completely.

Syntax:

```
DROP DATABASE database_name;
```

```
DROP TABLE table_name;
```

2. DATA MANIPULATION LANGUAGE (DML) + DATA QUERY LANGUAGE(DQL):

CRUD Operations:

The acronym **CRUD** refers to all of the major functions that need to be implemented in a relational database.

Operation	SQL	Description
CREATE DATA	INSERT INTO	Insert data into database
READ	SELECT	Retrieve/read data from database
UPDATE	UPDATE	Updates data from a database
DELETE	DELETE	Deletes data from a database

CRUD stands for CREATE(INSERT INTO), READ(SELECT), UPDATE, and DELETE statements in SQL.

2.1. INSERT:

INSERT command is used to insert data into the table.
It is possible to write the INSERT INTO statement in different ways given below:

2.1.1. Specify both the column names and the values to be inserted:
Syntax:

```
INSERT INTO table_name (column1, column2, column3,...) values(value1, value2, value3,...);
```

Example:

```
INSERT INTO employee
(emp_id, emp_name, salary, location)
VALUES
(1001, 'Aditya', 50000,'Mumbai');
```

2.1.2. You do not need to specify the column names in the SQL query If you are adding values for all the columns of the table,
however, make sure the order of the values is in the same order as the columns in the table.

Syntax:

```
INSERT INTO table_name values(value1, value2, value3,…);
```

Example:

```
INSERT INTO employee
VALUES
(1002,'Rahul', 45000,'Pune');
```

2.1.3. We can also insert multiple records using a single INSERT INTO query.

Syntax:

```
INSERT INTO table_name
(column1, column2, column3,…)
values(value1, value2, value3,…)
      (value1, value2, value3,…);
```

Example :

```
INSERT INTO employee (emp_id, emp_name, salary, location)
VALUES
(1003,'Rohit',40000,NULL),
(1004,'Saurav',60000,'Chennai'),
(1005,'Manish',30000,'Mumbai');
```

2.2. SELECT:

SELECT command is used to read data from database.

Syntax:

```
SELECT * FROM table_name;
```

Example:

SELECT * FROM employee;

Output:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

Here, * represents all columns from a table.

FROM clause is used to specify a data source, which can either be a table, view or a subquery.

To read specific columns from a table, use below syntax:

```
SELECT column1, column2 FROM table_name;
```

Example:

SELECT emp_id, emp_name, salary FROM employee;

Output:

Emp_id	emp_name	Salary
1001	Aditya	50000
1002	Rahul	45000
1003	Rohit	40000
1004	Saurav	60000
1005	Manish	30000

2.3. UPDATE:

The UPDATE statement is used to modify the existing data from a table.

Syntax:

```
UPDATE table_name
SET column_name1=new_value,
column_name2=new_value;
```

Example:
The following SQL query update the employee table with a new Location for all rows.
UPDATE employee SET location = Benguluru;

Table Before update:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

Table After update:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Benguluru
1002	Rahul	45000	Benguluru
1003	Rohit	40000	Benguluru
1004	Saurav	60000	Benguluru
1005	Manish	30000	Benguluru

CASE Statement:

The CASE statement goes through conditions and return a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result.

- If no conditions are true, it will return the value in the ELSE clause.
- If there is no ELSE part and no conditions are true, it returns NULL.
- Else, will have the original column value.Syntax:

```
UPDATE table_name SET column_name=
CASE column_name
WHEN condition THEN result1
WHEN condition THEN result2
ELSE result
END;
```

Example:

The following SQL query update the employee table with a different Location for all rows based on a condition.

The rows which does not satisfy any of these conditions, will have original location value.

```
UPDATE employee SET location=
CASE empid
WHEN 1001 THEN 'Chennai'
WHEN 1003 THEN 'Mumbai'
WHEN 1005 THEN 'Benguluru'
ELSE loc
END;
```

Table before update:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

Table after update:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Chennai
1002	Rahul	45000	Pune
1003	Rohit	40000	Mumbai
1004	Saurav	60000	Chennai
1005	Manish	30000	Benguluru

2.4. DELETE:

The DELETE statement is used to DELETE rows from a table.

Syntax:

```
DELETE FROM table_name;
```

Example:
The following DELETE statement will delete all rows from an employee table.

```
DELETE FROM employee;
```

Table after delete:

Emp_id	emp_name	Salary	Location

CONSTRAINTS:

SQL constraints are used to specify rules for the data in a table.

- SQL constraints 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 table.
- If there is any violation between the constraint and the data action, the action is aborted.

Constraint	Description
NOT NULL	Ensures that a NULL value can not be stored in a column
UNIQUE	This constraint makes sure that all the values in a column are different
CHECK	This constraint ensures that all the values in a column satisfy a specific condition
DAFAULT	This constraint consists of default value for a column when no value is specified
PRIMARY KEY	Ensures that the column value in every row is unique and has no null value
FOREIGN KEY	Helps to form parent child relationship between tables

The following constraints are commonly used in SQL:

1. **NOT NULL**

The NOT NULL constraint enforces a column to NOT accept NULL values.

Syntax:

```
CREATE TABLE Table_name
(
Column1 datatype NOT NULL,
Column2 datatype NOT NULL,
Column3 datatype,
...);
```

The following SQL ensures that the roll_no and address column will NOT accept NULL values when the "Student" table is created:

Example:

```
CREATE TABLE student
(
roll_no int NOT NULL,
stud_name varchar(20),
address varchar(25) NOT NULL );
```

To apply NOT NULL constraint on an existing database table, use below syntax

Syntax:

```
ALTER TABLE table_name MODIFY COLUMN column_name data_type  NOT NULL;
```

Example:

Below query is used to add a NOT NULL constraint on emp_id column of employee table:

```
ALTER TABLE employee MODIFY COLUMN emp_id int NOT NULL;
```

This constraint can be dropped using below syntax:

```
ALTER TABLE employee MODIFY COLUMN column_name data_type;
```

Example:

```
ALTER TABLE employee MODIFY COLUMN emp_id int ;
```

2. UNIQUE KEY

- The UNIQUE KEY constraint ensures that all values in a column are distinct.
- UNIQUE Constraint allows NULL values.
- Each table can have multiple UNIQUE Constraints.

Syntax:

```
CREATE TABLE table_name (
column1 datatype UNIQUE,
Column2 datatype UNIQUE,
Column3 datatype);
```

The following SQL ensures that the roll_no column will not accept duplicate values when the "Student" table is created:

Example:

```
CREATE TABLE student (
roll_no int UNIQUE,
stud_name varchar(20),
address varchar(25));
```

To apply UNIQUE KEY constraint on an existing database table, use below syntax.

Syntax:

```
ALTER TABLE table_name MODIFY COLUMN column_name data_type UNIQUE;
```

Example:

Below query is used to add a UNIQUE KEY constraint on emp_id column of employee table:

```
ALTER TABLE employee MODIFY COLUMN emp_id int UNIQUE;
```

This constraint can be dropped by below syntax:

```
ALTER TABLE employee DROP INDEX UNIQUE_KEY_COLUMN_NAME;
```

Example:

```
ALTER TABLE employee DROP INDEX emp_id;
```

3. PRIMARY KEY

- The PRIMARY KEY constraint is a combination of a NOT NULL and UNIQUE.
- PRIMARY KEY uniquely identifies each row in a table
- Primary keys must contain UNIQUE values, and cannot have NULL values.
  A table can have only ONE PRIMARY KEY but a single primary key can contain one or multiple columns

Syntax:

```
CREATE TABLE table_name
(
column1 datatype PRIMARY KEY,
column2 datatype,
column3 datatype,
…);
```

Example:

```
CREATE TABLE student
(roll_no int PRIMARY KEY,
stud_name varchar(20),
address varchar(25));
```

We can apply a PRIMARY KEY constraint on an existing database table using below syntax:

```
ALTER TABLE employee MODIFY COLUMN Column_name Data_type
PRIMARY KEY;
```

Example:

```
ALTER TABLE employee MODIFY COLUMN emp_id int PRIMARY KEY;
```

This constraint can be dropped by below syntax:

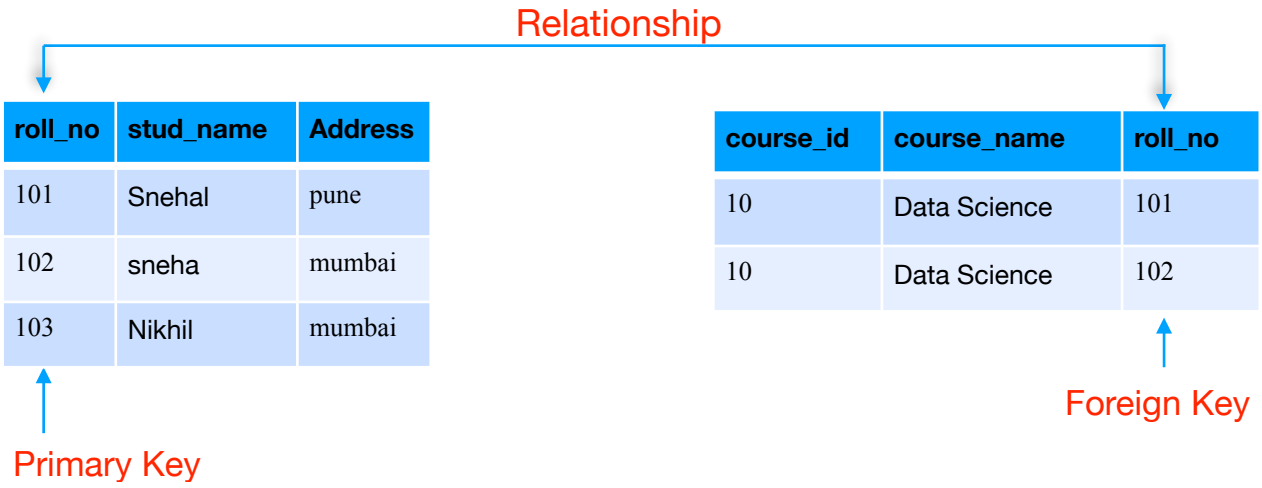
```
ALTER TABLE employee DROP PRIMARY KEY;
```

Example:

```
ALTER TABLE employee DROP PRIMARY KEY;
```

4. FOREIGN KEY

- A FOREIGN KEY is a field in one table, that refers to the PRIMARY KEY in another table.
- It is used to define a relation between two or multiple database tables.
- We can have more than one FOREIGN KEYs in a table .
- The table with the FOREIGN KEY is called the child or referencing table, and the table with the PRIMARY KEY is called the parent or referenced table.



Syntax:

```
CREATE TABLE child_table_name
(column1 datatype,
 column2 datatype,
 column3  datatype,
 FOREIGN KEY (column_of_child_table) references
parent_table(parent_table_primary_key)
ON DELETE CASCADE,
ON UPDATE CASCADE
);
```

Example:

Below query can be used to define roll_no as the FOREIGN KEY in course table which refers to the PRIMARY KEY of student table we have used above.

```
CREATE TABLE course
(course_id int,
course_name varchar(15),
Address varchar(25),
roll_no int,
FOREIGN KEY (roll_no) REFERENCES student(roll_no)
ON DELETE CASCADE
ON UPDATE CASCADE
);
```

The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, as it has to be one of the values contained in the parent table.

ON_DELETE_CASCADE : It is used with foreign key definition to cascade the parent table deletes in child table

ON_UPDATE_CASCADE: It is used with foreign key definition to cascade the parent table updates in child table

We can apply a FOREIGN KEY constraint on an existing database table using below syntax:

```
ALTER TABLE Table_name ADD FOREIGN KEY(child_table_column_name)
references Parent_table(Parent_Table_PK);
```

This constraint can be dropped by below syntax:

```
ALTER TABLE table_name DROP FOREIGN KEY Foreign_key_name;
```

Here, we can get the name of foreign key from create table statement .
To get the create table statement, use below syntax:

```
SHOW create table table_name;
```

5. CHECK

- The CHECK constraint is used to limit the value range that can be placed in a column. If you define a CHECK constraint on a column it will allow only certain values for this column.

Syntax:

```
CREATE TABLE table_name (column1 datatype, column2 datatype,
CHECK(Condition));
```

Example:
Below query will create a table that will restrict the roll_no column to have values > 100.

```
CREATE TABLE student
(roll_no int,
stud_name varchar(20),
address varchar(25),
CHECK(roll_no>100));
```

We can apply a Check constraint on an existing database table using below syntax:

```
ALTER TABLE Table_name MODIFY COLUMN Column_name data_type
CHECK(condition);
```

This constraint can be dropped by below syntax:

```
ALTER TABLE table_name DROP CONSTRAINT Check_constraint_name;
```

Here, we can get the name of constraint from create table statement .

6. DEFAULT

- The DEFAULT constraint is used to set a default value to a column.
- The default value will be added to all new records, if no other value is specified.

Syntax:

```
CREATE TABLE table_name
(column1 datatype DEFAULT  value,
column2 datatype,
column3 datatype);
```

Example:
Below query can be used to set a default value for address column as pune in student table

```
CREATE TABLE student
 (
Roll_no int,
stud_name varchar(20),
Address varchar(25) DEFAULT 'pune'
);
```

We can apply a DEFAULT constraint on an existing database table using below syntax:

```
ALTER TABLE table_name MODIFY COLUMN column_name data_type
DEFAULT value;
```

To drop default constraint ,use below syntax:

```
ALTER TABLE table_name ALTER column_name DROP DEFAULT;
```

7. AUTO_INCREMENT:

- The AUTO_INCREMENT attribute can be used to generate a unique identity for new rows.
- It is mandatory to define a column as key to auto_increment it .

Syntax:

```
CREATE TABLE table_name (column1 datatype primary key
Auto_increment, column2 datatype, column3 datatype);
```

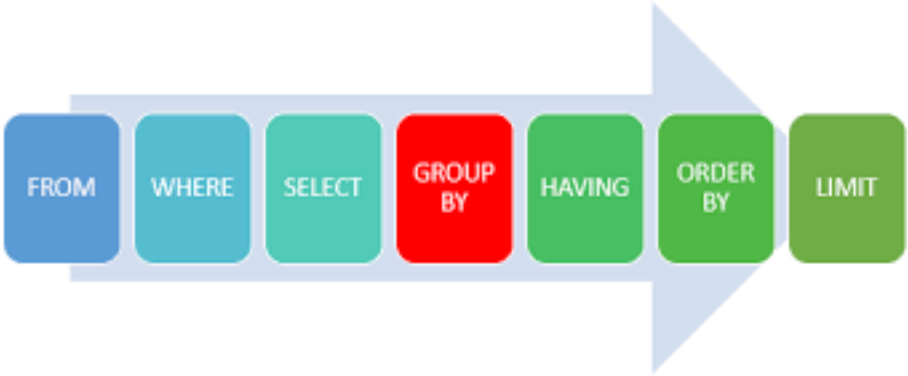
Example:

Below query can be used to apply auto_increment on roll_no column of Student table

```
CREATE TABLE student
 (
Roll_no int AUTO_INCREMENT PRIMARY KEY,
stud_name varchar(20),
Address varchar(25)
);
```

SQL CLAUSES:

The sequence of execution of sql clauses



1. **WHERE:**

- WHERE clause is used to filter the data.
- It helps us SELECT, UPDATE or DELETE only records that fulfils a specified condition.
- If we omit where clause , it will affect all rows of a table.

1.1. WHERE with SELECT:

WHERE clause can be used with SELECT statement to read particular rows satisfying the filter condition

Syntax:

```
SELECT * FROM table_name where condition;
```

Example:

Below statement can be used to read details about an employee with emp_id=1001.

```
SELECT * FROM employee WHERE emp_id=1001;
```

Output:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai

1.2. WHERE with UPDATE:

WHERE clause can be used with UPDATE statement to UPDATE only records satisfying a filter condition.

Syntax:

```
UPDATE table_name SET column_name=new_value where condition;
```

Example:

Below statement will UPDATE the location value where it is NULL:

```
UPDATE employee SET location='Chennai' WHERE Location IS NULL;
```

Table before update:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

Table after update:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	**Chennai**
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

1.3. WHERE with DELETE:

WHERE clause can be used with DELETE statement to DELETE only specific records.

Syntax:

```
DELETE FROM table_name where  condition;
```

Example: Below statement is used to DELETE a record where emp_id is 1005:

```
DELETE FROM employee WHERE emp_id=1005;
```

Table before delete:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

Table after delete:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai

The difference between DELETE and TRUNCATE here is,
We can delete specific records using delete with where condition.
Whereas, truncate can only be used to empty the table .you cannot use the where condition with truncate.

• OPERATORS IN SQL:

WHERE clause can be combined with the below operators to fetch the required result in SQL

Operator	Description
=	Equal To
!=	Not Equal To
>	Greater than
<	Less than
>=	Greater than equal to
<=	Less than equal to
BETWEEN	Between an inclusive range
IN	Equal to multiple values
LIKE	Search for a pattern
IS NULL	Checks for NULL values
AND	Used to combine multiple where conditions
OR	Used to combine multiple where conditions
LIKE	Used to specify a pattern

1. OR and AND

The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND are TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.

1.1. OR

It is a SQL statement used to access records which satisfies either condition combined using OR.

Syntax:

```
SELECT column1, column2  from table_name where Condition1 OR Condition 2;
```

Example:

The following SQL statement selects all fields from employee where emp_id=1001 or emp_name = Saurav

```
SELECT * FROM employee WHERE emp_id= 1001 OR emp_name='Saurav';
```

Output:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1004	Saurav	60000	Chennai

1.2. AND

It is a SQL statement used to access records which satisfies both condition

Syntax :

```
SELECT column1, column2 FROM table_name WHERE Condition1 AND Condition 2;
```

Example :

```
SELECT * FROM employee WHERE emp_id=1001 AND emp_name = Aditya;
```

Output :

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai

2. LIKE

- The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
- There are two wildcards often used with the LIKE operator:
  The percent sign (%) represents zero, one, or multiple characters
  The underscore sign (_) represents one, single character

Syntax:

```
SELECT * FROM table_name WHERE column LIKE pattern;
```

Example1: query to fetch all rows WHERE emp_name starts with R.

```
SELECT * FROM employee WHERE emp_name LIKE 'R%';
```

Output :

Emp_id	emp_name	Salary	Location
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL

Example 2: SELECT all records WHERE the value of the emp_name column does NOT start with the letter "A".

Syntax :

```
SELECT * FROM table_name WHERE column NOT LIKE pattern;
```

Example3: Query to display records where emp_name is not having A

```
SELECT * FROM employee WHERE emp_name NOT LIKE 'A%';
```

Output :

Emp_id	emp_name	Salary	Location
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

Example4: select all records where the value of the column starts with letter ‘a’ and ends with the letter ‘a'

Example:

```
SELECT * FROM employee WHERE emp_name LIKE 'a%a';
```

Output :

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai

Example 5: SELECT all records where the emp_name contains the letter "a".

```
SELECT * FROM employee WHERE emp_name LIKE '%a%';
```

Output:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	mumbai
1002	Rahul	45000	Pune
1004	Saurav	60000	Chennai
1005	Manish	30000	mumbai

3. IN and NOT IN

These operators allows you to specify one or multiple values in a WHERE clause.
Syntax:

```
SELECT * FROM table_name
WHERE Column IN/NOT IN (Value1, Value2, Value3...)
```

Example1 :
The following SQL statement selects all rows where emp_name is in "Rohit", “Saurav":

```
SELECT * FROM employee WHERE emp_name IN ('Rohit','Saurav');
```

Output:

Emp_id	emp_name	Salary	Location
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai

Example2:

The following SQL statement selects all employees except "Rohit", "Saurav":

Example :

```
SELECT * FROM employee WHERE emp_name NOT IN (Rohit, Saurav);
```

Output:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1005	Manish	30000	Mumbai

4. BETWEEN

- The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.
- Here, start and end values are included in the output.

Syntax :

```
SELECT * FROM table_name WHERE Column_name BETWEEN Value1 AND Value2;
```

Example :

```
SELECT * FROM employee WHERE salary BETWEEN 40000 AND 50000;
```

Output:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL

NOT BETWEEN Example :

To display the records outside the given range , use NOT BETWEEN:

Syntax :

```
SELECT * FROM table_name WHERE Column_name NOT BETWEEN Value1
AND Value2;
```

Example :

```
SELECT * FROM employee WHERE salary NOT BETWEEN 40000 AND 50000;
```

Output :

Emp_id	emp_name	Salary	Location
1004	Saurav	60000	Chennai
1005	Manish	30000	Mumbai

2. DISTINCT :

DISTINCT clause is used to fetch the UNIQUE values from a column of a table.

Syntax:

```
SELECT DISTINCT column_name FROM table_name;
```

Example: Below query can be used to fetch UNIQUE location values FROM employee table

```
SELECT DISTINCT location FROM employee;
```

Output:

Location
Mumbai
Pune
NULL
Chennai

To fetch unique rows from a table, use below syntax:

```
SELECT DISTINCT * FROM table_name;
```

3. LIMIT:

LIMIT clause is used to limit the number of records to return.

Syntax:

```
SELECT * FROM table_name LIMIT  OFFSET, NUMBER OF ROWS
```

Here,
OFFSET parameter specifies the number of rows to skip from top.
NUMBER OF ROWS parameter specifies the number of rows to read.

Example:
Below statement will skip 2 rows from top and return next 2 rows in the output.

```
SELECT * FROM employee LIMIT 2,2;
```

Output:

Emp_id	emp_name	Salary	Location
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai

We can also use limit clause like below:

```
SELECT * FROM employee LIMIT 2;
```

Here, when we pass only one parameter to limit clause, it is the number of rows to read. OFFSET value becomes 0 in this case. This statement will read top 2 rows from a table

Output:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1002	Rahul	45000	Pune

4. ORDER BY :

- ORDER BY clause is used to sort the data in an ascending or descending order.
- The ORDER BY keyword sorts the records in ascending order by default.
- To sort the records in descending order, use the DESC keyword.

Syntax:

```
SELECT * FROM table_name ORDER BY column1, column2;
```

Example: Below statement will sort the data of employee table in ascending order of emp_name:

```
SELECT * FROM employee ORDER BY emp_name;
```

Output:

Emp_id	emp_name	Salary	Location
1001	Aditya	50000	Mumbai
1005	Manish	30000	Mumbai
1002	Rahul	45000	Pune
1003	Rohit	40000	NULL
1004	Saurav	60000	Chennai

Below statement will sort the data of employee table in descending order of emp_name:

```
SELECT * FROM employee ORDER BY emp_name DESC;
```

Output:

Emp_id	emp_name	Salary	Location
1004	Saurav	60000	Chennai
1003	Rohit	40000	NULL
1002	Rahul	45000	Pune
1005	Manish	30000	Mumbai
1001	Aditya	50000	Mumbai

ALIAS :

- SQL aliases are used to give a table, or a column in a table, a temporary name. Aliases are often used to make column names more readable.
- An alias only exists for the duration of that query. An alias is created with the AS keyword.

Syntax to use alias for a column name :

```
SELECT column_name AS alias_name FROM table_name;
```

Syntax to use alias for a table name:

```
SELECT column_name(s) FROM table_name AS alias_name;
```

Example:

The following SQL statement creates two aliases, one for the calculated column and one for the table:

```
SELECT emp_id, emp_name, salary, 0.25*salary as per_salary FROM employee as e;
```

Output:

Emp_id	emp_name	Salary	per_salary
1001	Aditya	50000	12500.00
1002	Rahul	45000	11250.00
1003	Rohit	40000	10000.00
1004	Saurav	60000	15000.00
1005	Manish	30000	7500.00

- **AGGREGATE FUNCTIONS:**
- An aggregate function performs calculations on a set of values and returns a single value.

1. MIN(): It returns minimum value in the selected column except null
2. MAX(): Will returns Maximum value in the selected column except null
3. SUM(): Sum of all Non-Null Values of a numeric column.
4. AVG(): This function returns average value of a numeric column - It will apply for Non-Null Values

Syntax:

```
SELECT MIN(column name),MAX(column name),SUM(column name),AVG(column name) FROM table_name;
```

Example:

```
SELECT MAX(salary),MIN(salary),SUM(salary),AVG(salary)  FROM employee;
```

Output:

Max(salary)	Min(salary)	sum(salary)	Avg(salary)
60000	30000	225000	45000.0000

5. COUNT():

- Returns total number of records when used with * or any constant number
- Returns total number of Non-Null Values of a column when used with a specific column

Syntax:

```
SELECT COUNT(*) FROM Table_name;
SELECT COUNT(1) FROM Table_name;
SELECT COUNT(column_name)  FROM table_name;
```

Example:

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

Output:

count(salary)
5

5. GROUP BY :

- The GROUP BY statement groups rows that have same values into summary rows.
- The GROUP BY statement is often used with aggregate functions like (COUNT(), MAX(), MIN(), SUM(), AVG()
- It returns a single row for each group

Syntax:

```
SELECT
Column,
AGGREGATE FUNCTION(Column)
FROM table_name
WHERE condition
GROUP BY Column;
```

Example: Below query will create groups based on location values and will calculate the number of employee, sum of salary, avg of salary for each group

```
SELECT location, count(emp_id), sum(salary) , avg(salary) FROM employee GROUP BY location;
```

Output:

Location	count(emp_id)	sum(salary)	avg(salary)
Mumbai	2	80000	40000.0
Pune	1	45000	45000.0
NULL	1	40000	40000.0
Chennai	1	60000	60000.0

6. HAVING :

- The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
- It is used with group by and aggregate functions to filter out grouped data.

Syntax :

```
SELECT
Column,
AGGREGATE FUNCTION(Column)
FROM table_name
WHERE condition
GROUP BY Column
HAVING Condition;
```

Example: Below query will fetch locations where employee count>1

SELECT location, count(emp_id) FROM employee GROUP BY location HAVING count(emp_id)>1

Output:

Location	Count(emp_id)
Mumbai	2

7. UNION and UNION ALL:

These are used to append the data from different SELECT statements.

- Every SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in every SELECT statement must also be in the same order

Syntax:

```
SELECT col_name FROM table_name
Union/union all
SELECT col_name FROM table_name;
```

Student table:

roll_no	stud_name	Address
101	Snehal	pune
102	sneha	mumbai
103	Nikhil	mumbai
104	nikita	pune
105	Shubham	pune

Course table:

course_id	course_name	roll_no
10	Data Science	101
10	Data Science	102
20	java	106
20	java	107

Example: below statement will append the result of both select queries

```
SELECT roll_no FROM student
union all
SELECT roll_no FROM course;
```

Output:

roll_no
101
102
103
104
105
101
102
106
107

below statement will append the result of both select queries.
Union will remove the duplicate values

```
SELECT roll_no FROM student
union
SELECT roll_no FROM course;
```

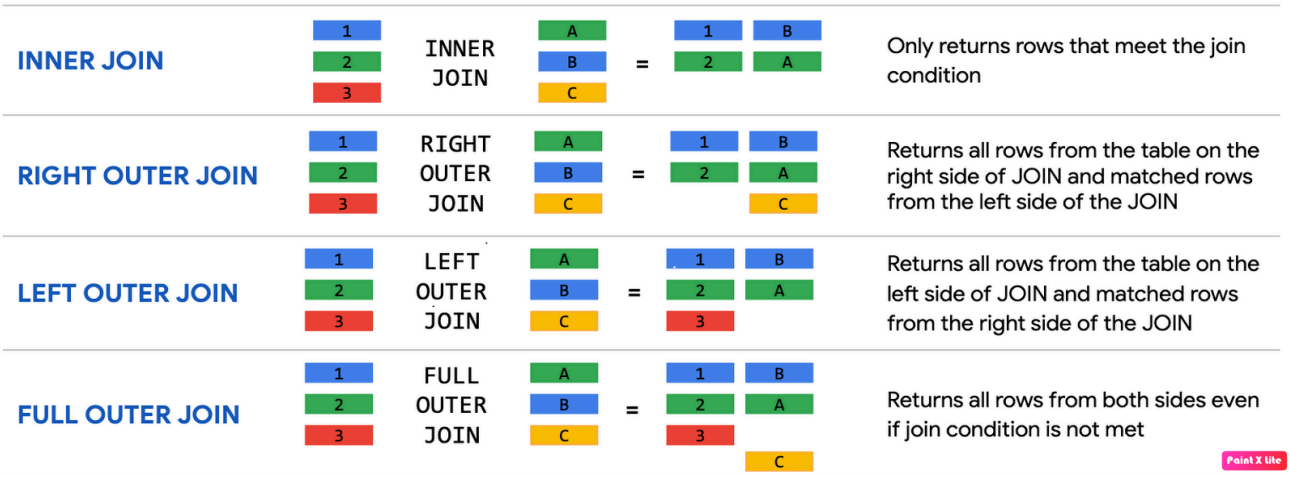
Output:

Roll_no
101
102
103
104
105
106
107

• SQL JOINS

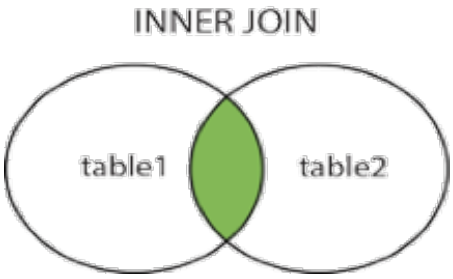
- Joins are used to combine the data from two or more database tables
- Join is an SQL statement it is used to join two or more tables based on the column having similar values in the tables being joined.

Types of JOINS



1. INNER JOIN

- The INNER JOIN keyword selects records that have matching values in both tables.
- Inner join is a SQL statement which is use to show records in columns which are related to common values



Syntax :

```
SELECT column_name(s)
FROM  table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

NOTE : Refer student and course table used above.
Below statement can be used to join student table with course table on the basis of roll_no values, which will give you common students present in both the tables.

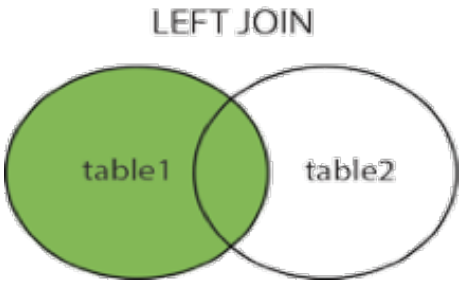
```
SELECT student. roll_no, stud_name, address, course_id, course_name
FROM
student
INNER JOIN
course
ON student.roll_no=course.roll_no;
```

Output:

roll_no	stud_name	Address	course_id	course_name
101	Snehal	pune	10	Data science
102	sneha	mumbai	10	Data science

2. LEFT JOIN/ LEFT OUTER JOIN

- The LEFT JOIN keyword returns all records from the LEFT table (table1), and the matching records from the right table (table2).
- The result is 0 records FROM the right side, if there is no match.



Syntax:

```
SELECT column_name(s)
FROM table1 LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

Example :

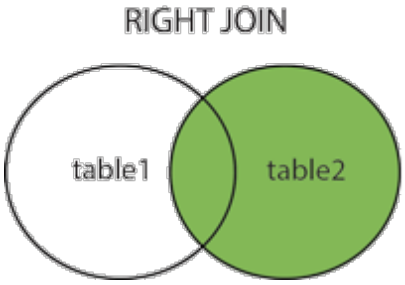
```
SELECT student.roll_no, stud_name, address, course_id, course_name
FROM
student
LEFT JOIN
course
ON student.roll_no=course.roll_no;
```

Output:

roll_no	stud_name	Address	Course_id	Course_name
101	Snehal	pune	10	Data science
102	sneha	mumbai	10	Data science
103	Nikhil	mumbai	NULL	NULL
104	nikita	pune	NULL	NULL
105	Shubham	pune	NULL	NULL

3. RIGHT JOIN / RIGHT OUTER JOIN

- The RIGHT JOIN keyword ,returns all records from the RIGHT table(Table2) and the matching records from the LEFT table (table1).
- The result is 0 records FROM the left side, if there is no match.



Syntax :

```
SELECT column_name(s)
FROM table1 RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

Example :

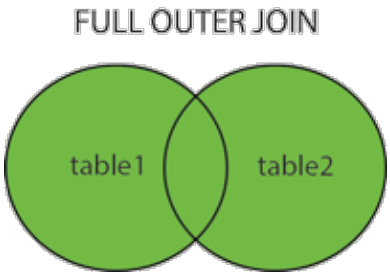
```
SELECT student.roll_no, stud_name, address, course_id, course_name
FROM
student
RIGHT JOIN
course
ON student.roll_no=course.roll_no;
```

Output :

roll_no	stud_name	Address	course_id	course_name
101	Snehal	Pune	10	Data science
102	Sneha	Mumbai	10	Data science
NULL	NULL	NULL	20	java
NULL	NULL	NULL	20	java

4. FULL JOIN / FULL OUTER JOIN

- MySQL does not support FULL JOIN, so you have to combine RIGHT JOIN, UNION and LEFT JOIN to get an equivalent.



Syntax :

```
SELECT column_name(s)
FROM table1 LEFT JOIN table2
ON table1.column_name = table2.column_name
UNION
SELECT column_name(s)
FROM table1 RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

Example :

```
select s.roll_no, stud_name, address, course_id, course_name
from student s
right join
course c
on s.roll_no=c.roll_no
union
select s.roll_no, stud_name, address, course_id, course_name
from student s
left join
course c
on s.roll_no=c.roll_no;
```

Output :

Roll_no	stud_name	Address	course_id	course_name
101	Snehal	pune	10	data science
102	sneha	mumbai	10	data science
NULL	NULL	NULL	20	java
103	Nikhil	mumbai	NULL	NULL
104	nikita	pune	NULL	NULL
105	Shubham	pune	NULL	NULL

SUBQUERIES:

A Subquery is an inner query that is placed within an outer SQL query using different SQL clauses like WHERE and FROM.

Types of subqueries:

1. **Scalar Subqueries:** the type of subqueries where inner query returns only single value

1.1. Scalar subquery in WHERE clause:

Syntax:

```
SELECT * FROM table_name WHERE col_name=(inner query)
```

Example:
Below query will fetch the details of an employee having max salary

```
SELECT * FROM employee WHERE salary=(SELECT max(salary) FROM employee);
```

Output:

Emp_id	emp_name	Salary	Location
1004	Saurav	60000	Chennai

1.2. Scalar subquery in FROM clause:
When we use a subquery with FROM clause, the result set returned is considered as a temporary table and thus it is mandatory to give an alias to this derived table.
Syntax:

```
SELECT
column1, column2 FROM
(select column from table_name where condition ) SQ Where condition;
```

Example: Fetch the name and salary of an employee having salary equal to avg salary

```
SELECT emp_id, emp_name, salary, sal
FROM employee e
Inner join
(SELECT avg(salary) as sal from employee ) as avg_sal
On e.salary=avg_sal.sal;
```

Here, AVG_SAL is the alias given to the derived table or an inner query. It is mandatory when we use query in a from clause.

Output:

Emp_id	emp_name	Salary	Sal
1002	Rahul	45000	45000.0000

2. Multi-row subquery : The type of subquery where inner query returns more than one value

Syntax:

```
SELECT * FROM table_name WHERE column_name IN/NOT IN (inner query)
```

Example: 1. Fetch roll_no from student which are not present in course table

```
SELECT roll_no, stud_name, address
FROM student s
where roll_no NOT IN (
SELECT distinct roll_no  from course) ;
```

Output:

roll_no	stud_name	Address
103	Nikhil	Mumbai
104	Nikita	Pune
105	Shubham	Pune

2. Calculate the count of records returned from union all query

```
SELECT COUNT(*) as total_rows
FROM
(SELECT roll_no FROM student
union all
SELECT roll_no FROM course) A;
```

Here, A is an alias for a subquery

Output:

Total_rows
9

ANALYTICAL FUNCTIONS:

- A window function performs a calculation across a set of table rows that are somehow related to the current row.
- This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities.

Sample table emp:

emp_id	emp_name	dept_name	Salary
101	Mohan	Admin	4000
102	Rahul	HR	3000
103	Akbar	IT	4000
104	Dorvin	Finance	6500
105	Rohit	HR	3000
106	Rajesh	Finance	5000
107	Preet	HR	7000
108	Maryam	Admin	4000
109	Sanjay	IT	6500
110	Vasudha	IT	7000
111	Melinda	IT	8000
112	Komal	IT	10000
113	Gautham	Admin	2000
114	Manisha	HR	3000
115	Chandni	IT	4500
116	Satya	Finance	6500
117	Adarsh	HR	3500
118	Tejaswi	Finance	5500
119	Cory	HR	8000
120	Monica	Admin	5000

1. ROW_NUMBER():

It is an analytical function used to return a sequential number to each row of the partition

OVER: It is an helping clause used with window functions to define partition and sort order
PARTITION_BY: Used to divide data into partitions and perform computation on each subset of partitioned data.
ORDER BY: Used to sort the data within defined partition
Syntax:

```
SELECT column,
ROW_NUMBER() OVER
(PARTITION BY column name
ORDER BY column name
) FROM table_name;
```

Example: Display oldest 2 employees from each department(consider min emp_id is the oldest emp)

```
select * from(
select *,
row_number() over (partition by dept_name order by emp_id) as rn from emp
) SQ
where rn<3;
```

Here,SQ is an alias which used for the inner query
rn is al alias used for the column calculated using row_number() function.

Output:

Empid	emp_name	dept_name	Salary	Rn
101	Mohan	Admin	4000	1
108	Maryam	Admin	4000	2
104	Dorvin	Finance	6500	1
106	Rajesh	Finance	5000	2
102	Rahul	HR	3000	1
105	Rohit	HR	3000	2
103	Akbar	IT	4000	1
109	Sanjay	IT	6500	2

2. LEAD() AND LAG ():

- It can often be useful to compare rows to preceding or previous rows.
- You can use LAG or LEAD to create columns that pull values from other rows—all you need to do is enter which column to pull from and how many rows away you would like to do the pull.
- LAG pulls from previous rows and LEAD pulls from preceeding rows:

expression	It is a **column name** or any built-in function whose value return by the function.
offset	It contains the number of rows succeeding from the current row. It should be a **positive integer** value. If it is **zero**, the function evaluates the result for the current row. If we **omit** this, the function uses 1 by default.
default_value	It is a value that will return when we have no subsequent row from the current row. If we omit this, the function returns the **null value**.
OVER	OVER It is responsible for partitioning rows into groups. If it is **empty**, the function performs an operation using all rows.
PARTITION BY	It split the rows in the result set into partitions to which a function is applied. If we have not specified this clause, all rows treated as a single row in the result set.
ORDER BY	It determines the sequence of rows in the partitions before the function is applied.

LEAD:

```
SELECT column,
LEAD(expression, offset, default_value) OVER
(PARTITION BY column name
ORDER BY column name
) FROM table_name;
```

LAG:

```
SELECT column,
LAG(expression, offset, default_value) OVER
(PARTITION BY column name
ORDER BY  column name
) FROM table_name;
```

Example :

```
SELECT  *,
LEAD(salary,1,0) over () as next_sal,
LAG(salary,1,0) over () as prev_sal from emp;
```

Output:

Empid	emp_name	dept_name	Salary	next_sal	prev_sal
101	Mohan	Admin	4000	3000	0
102	Rahul	HR	3000	4000	4000
103	Akbar	IT	4000	6500	3000
104	Dorvin	Finance	6500	3000	4000
105	Rohit	HR	3000	5000	6500
106	Rajesh	Finance	5000	7000	3000
107	Preet	HR	7000	4000	5000
108	Maryam	Admin	4000	6500	7000
109	Sanjay	IT	6500	7000	4000
110	Vasudha	IT	7000	8000	6500
111	Melinda	IT	8000	10000	7000
112	Komal	IT	10000	2000	8000
113	Gautham	Admin	2000	3000	10000
114	Manisha	HR	3000	4500	2000
115	Chandni	IT	4500	6500	3000
116	Satya	Finance	6500	3500	4500
117	Adarsh	HR	3500	5500	6500
118	Tejaswi	Finance	5500	8000	3500
119	Cory	HR	8000	5000	5500
120	Monica	Admin	5000	0	8000

3. RANK() and DENSE_RANK():

It is mandatory to sort the data within partition to assign rank and dense_Rank to them

RANK : Rank of current row within its partition ,with gap

```
SELECT column,
RANK() OVER
(PARTITION BY column name
ORDER BY column name
) FROM table_name;
```

DENSE_RANK : Rank of current row within its partition, without gaps

```
SELECT column,
DENSE_RANK() OVER
(PARTITION BY column name
ORDER BY column name
) FROM table_name;
```

Example :

```
SELECT emp_id, dept_name, salary,
rank() OVER (PARTITION BY dept_name ORDER BY salary DESC) as rnk
,dense_rank() OVER (PARTITION BY dept_name ORDER BY salary DESC) as dense_rnk
FROM emp;
```

Output:

emp_id	dept_name	Salary	rnk	dense_rnk
120	Admin	5000	1	1
101	Admin	4000	2	2
108	Admin	4000	2	2
113	Admin	2000	4	3
104	Finance	6500	1	1
116	Finance	6500	1	1
118	Finance	5500	3	2
106	Finance	5000	4	3

119	HR	8000	1	1
107	HR	7000	2	2
117	HR	3500	3	3
102	HR	3000	4	4
105	HR	3000	4	4
114	HR	3000	4	4
112	IT	10000	1	1
111	IT	8000	2	2
110	IT	7000	3	3
109	IT	6500	4	4
115	IT	4500	5	5
103	IT	4000	6	6