MYSQL

1. What is MySQL?

SQL (Structured Query Language) is a standardized language used for querying, updating, and managing relational databases. It allows users to interact with the database through commands to perform tasks like querying data, inserting new records, updating existing data, and managing database schema. MySQL is a popular open-source relational database management system (RDBMS) that uses Structured Query Language (SQL) for accessing and managing data. It is widely used in web applications and is known for its reliability and performance.

2. 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 Syntax:

SQL keywords are NOT case sensitive: select is the same as SELECT

Semicolon after SQL Statements?

- Some database systems require a semicolon at the end of each SQL statement. Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

Some of The Most Important SQL Commands

- SELECT - extracts data from a database
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts new data into a database
- CREATE DATABASE - creates a new database
- ALTER DATABASE - modifies a database
- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table
- DROP TABLE - deletes a table
- CREATE INDEX - creates an index (search key)
- DROP INDEX - deletes an index

How to create a database in cmd?

1. Install XAMPP

1. **Download XAMPP**: If you haven't already, download the XAMPP installer from the official XAMPP website.
2. **Install XAMPP**: Run the installer and follow the prompts. By default, XAMPP will be installed in C:\xampp.

2. Start XAMPP Services

1. **Open XAMPP Control Panel**: Go to the XAMPP installation directory (C:\xampp) and open xampp-control.exe. Alternatively, you can find XAMPP in your Start menu.
2. **Start Apache and MySQL**: In the XAMPP Control Panel, click the "Start" buttons next to **Apache** and **MySQL**. This will start the web server and database server.

3. Access MySQL Command Line

1. **Open Command Prompt**: Press Win + R, type cmd, and press Enter.
2. **Navigate to the MySQL Bin Directory**:
   [cd C:\xampp\mysql\bin]
3. **Login to MySQL**: Use the MySQL command-line client to log in. By default, the root user does not have a password.

```
C:\xampp\mysql\bin>mysql -u root -p -h localhost
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 189
Server version: 10.4.32-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

4. Create a Database

1. **Create Database**: Once logged in, you can create a new database using SQL commands
   For creating database, the basic syntax is "CREATE DATABASE database_name;"
   Example:

```
MariaDB [(none)]> create database example_database;
Query OK, 1 row affected (0.008 sec)
```

5.Verify Creation: To ensure the database was created, you can list all databases:

```
MariaDB [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| class              |
| example_database   |
| information_schema |
| mydatabase         |
| mysql              |
| new_database       |
| performance_schema |
| phpmyadmin         |
| test               |
+--------------------+
9 rows in set (0.075 sec)
```

Example_database

6. Use the Database: To start working with your new database, use the USE command:

```
MariaDB [(none)]> use example_database;
Database changed
MariaDB [example_database]>
```

7. Create Tables and Add Data: Define a table structure within your database.

By creating table we should add create table and add columns with example like first name, last name, email, obtained marks etc…[add data which is required]

```
MariaDB [example_database]> create table example(
    -> First_name varchar (50) not null,
    -> last_name varchar (50) not null,
    -> obtained_marks decimal (10,2)
    -> );
Query OK, 0 rows affected (0.089 sec)
```

8. Insert Data: Add data to your table:

```
MariaDB [example_database]> insert into example (First_name,last_name,obtained_marks)
    -> values('raju','p',98);
Query OK, 1 row affected (0.020 sec)

MariaDB [example_database]> insert into example (First_name,last_name,obtained_marks)
    -> values('rani','s',90);
Query OK, 1 row affected (0.041 sec)
```

9. Query Data: Retrieve data from your table

General syntax SELECT * FROM table_name;

```
MariaDB [example_database]> select * from example;
+------------+-----------+-----------------+
| First_name | last_name | obtained_marks  |
+------------+-----------+-----------------+
| raju       | p         |           98.00 |
| rani       | s         |           90.00 |
+------------+-----------+-----------------+
2 rows in set (0.040 sec)
```

1. Basic SQL Queries

SELECT: Retrieve data from one or more tables.

SELECT column1, column2

FROM table_name;

```
MariaDB [example_database]> select First_name,last_name,obtained_marks
    -> from example;
+------------+-----------+----------------+
| First_name | last_name | obtained_marks |
+------------+-----------+----------------+
| raju       | p         |          98.00 |
| rani       | s         |          90.00 |
+------------+-----------+----------------+
2 rows in set (0.001 sec)
```

2. Filtering Data

a. **WHERE**: Filter records based on specific conditions.

The WHERE clause is used to filter records.

```
MariaDB [example_database]> select First_name,last_name
    -> from example
    -> where obtained_marks = 98;
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id:    188
Current database: example_database

+------------+-----------+
| First_name | last_name |
+------------+-----------+
| raju       | p         |
+------------+-----------+
1 row in set (0.025 sec)
```

b. **AND, OR, NOT**: Combine multiple conditions.

- **And**: The AND operator is used to filter records based on more than one condition, like if you want to return all example(table name) from obtained marks '90' and last name=s:

```
MariaDB [example_database]> select First_name,last_name
    -> from example
    -> where obtained_marks=90 and last_name='s';
+------------+-----------+
| First_name | last_name |
+------------+-----------+
| rani       | s         |
+------------+-----------+
1 row in set (0.009 sec)
```

- **Using OR to Combine Conditions**

SELECT first_name, last_name
FROM employees
WHERE department = 'example' OR department = 'example';

```
MariaDB [example_database]> select First_name,last_name
    -> from example
    -> where last_name='s' or obtained_marks=50;
+------------+-----------+
| First_name | last_name |
+------------+-----------+
| rani       | s         |
| rajesh     | s         |
+------------+-----------+
2 rows in set (0.038 sec)
```

- **Using NOT to Exclude Conditions**

  SELECT first_name, last_name
  FROM example
  WHERE NOT column name = data;

```
MariaDB [example_database]> SELECT First_name, last_name, obtained_marks
    -> FROM example
    -> WHERE NOT last_name = 's';
+------------+-----------+----------------+
| First_name | last_name | obtained_marks |
+------------+-----------+----------------+
| raju       | p         |          98.00 |
| sangeetha  | m         |          77.00 |
+------------+-----------+----------------+
2 rows in set (0.002 sec)
```

Advanced filtering options

a. 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 in conjunction with the LIKE operator:

- The percent sign % represents zero, one, or multiple characters
- The underscore sign _ represents one, single character
- Syntax:- SELECT *column1, column2, ...*
  FROM *table_name*
  WHERE *column_Name* LIKE *pattern;*

```
MariaDB [example_database]> SELECT First_name, last_name
    -> FROM example
    -> WHERE First_name LIKE 'r%';
+------------+-----------+
| First_name | last_name |
+------------+-----------+
| raju       | p         |
| rani       | s         |
| rajesh     | s         |
+------------+-----------+
3 rows in set (0.039 sec)
```

b. BETWEEN: The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. The BETWEEN operator is inclusive: begin and end values are included.

Syntax: SELECT column1, column2
FROM table_name
WHERE column_name BETWEEN values1 AND values2;

```
MariaDB [example_database]> select First_name,last_name,obtained_marks
    -> from example
    -> where obtained_marks between 50 and 100;
+------------+-----------+----------------+
| First_name | last_name | obtained_marks |
+------------+-----------+----------------+
| raju       | p         |          98.00 |
| rani       | s         |          90.00 |
| rajesh     | s         |          67.00 |
| sangeetha  | m         |          77.00 |
+------------+-----------+----------------+
4 rows in set (0.047 sec)
```

NOT BETWEEN

To display the products outside the range of the previous example, use NOT BETWEEN:

```
MariaDB [example_database]> select First_name,last_name,obtained_marks
    -> from example
    -> where obtained_marks not between 20 and 80;
+------------+-----------+----------------+
| First_name | last_name | obtained_marks |
+------------+-----------+----------------+
| raju       | p         |          98.00 |
| rani       | s         |          90.00 |
+------------+-----------+----------------+
2 rows in set (0.001 sec)
```

c. Using IN: The IN operator allows you to specify multiple values in a WHERE clause.
The IN operator is a shorthand for multiple OR conditions.

syntax: SELECT column1, column2
FROM table_name
WHERE column IN (value1, value2, ...);

```
MariaDB [example_database]> select First_name,last_name,obtained_marks
    -> from example
    -> where last_name in ('s','p');
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id:    264
Current database: example_database

+------------+-----------+----------------+
| First_name | last_name | obtained_marks |
+------------+-----------+----------------+
| raju       | p         |          98.00 |
| rani       | s         |          90.00 |
| rajesh     | s         |          67.00 |
+------------+-----------+----------------+
3 rows in set (0.053 sec)
```

3. Sorting and Limiting Data

ORDER BY: Sort the result set.

The ORDER BY clause is used to sort the result set of a query. By default, ORDER BY sorts the data in ascending order (ASC). If you want the data to be sorted in descending order, you can specify DESC.

Syntax: SELECT column1, column2
FROM table_name
ORDER BY column1 [ASC|DESC];

```
MariaDB [example_database]> SELECT First_name, last_name
    -> FROM example
    -> ORDER BY last_name ASC;
+------------+-----------+
| First_name | last_name |
+------------+-----------+
| sangeetha  | m         |
| raju       | p         |
| rani       | s         |
| rajesh     | s         |
+------------+-----------+
4 rows in set (0.001 sec)
```

```
MariaDB [example_database]> SELECT First_name, last_name
    -> FROM example
    -> ORDER BY last_name DESC;
+------------+-----------+
| First_name | last_name |
+------------+-----------+
| rani       | s         |
| rajesh     | s         |
| raju       | p         |
| sangeetha  | m         |
+------------+-----------+
4 rows in set (0.001 sec)
```

LIMIT: Limit the number of rows returned.

The LIMIT clause restricts the number of rows returned by the query. This is particularly useful when you want to retrieve only a subset of results from a potentially large result set.

SELECT column1, column2
FROM table_name
LIMIT number;

```
MariaDB [example_database]> SELECT First_name, last_name
    -> FROM example
    -> limit 4;
+------------+-----------+
| First_name | last_name |
+------------+-----------+
| raju       | p         |
| rani       | s         |
| rajesh     | s         |
| sangeetha  | m         |
+------------+-----------+
4 rows in set (0.001 sec)

MariaDB [example_database]> SELECT First_name, last_name
    -> FROM example
    -> limit 10;
+------------+-----------+
| First_name | last_name |
+------------+-----------+
| raju       | p         |
| rani       | s         |
| rajesh     | s         |
| sangeetha  | m         |
+------------+-----------+
4 rows in set (0.000 sec)

MariaDB [example_database]> SELECT First_name, last_name
    -> FROM example
    -> limit 2;
+------------+-----------+
| First_name | last_name |
+------------+-----------+
| raju       | p         |
| rani       | s         |
+------------+-----------+
```

Common Aggregation Functions

1. **COUNT**
    - Purpose: Counts the number of rows in a dataset or the number of non-NULL values in a specific column.
    - **Syntax: COUNT(expression)**

```
MariaDB [example_database]> SELECT COUNT(*) FROM example;
+----------+
| COUNT(*) |
+----------+
|        7 |
+----------+
1 row in set (0.001 sec)
```

2. **SUM**

- Purpose: Calculates the total sum of a numeric column.
- **Syntax: SUM(column_name)**

```
MariaDB [example_database]> SELECT sum(obtained_marks) FROM example;
+---------------------+
| sum(obtained_marks) |
+---------------------+
|              473.00 |
+---------------------+
1 row in set (0.038 sec)
```

3. **AVG**

- Purpose: Computes the average value of a numeric column.
- **Syntax:** AVG(column_name)

```
MariaDB [example_database]> SELECT AVG(obtained_marks) FROM example;
+---------------------+
| AVG(obtained_marks) |
+---------------------+
|           67.571429 |
+---------------------+
1 row in set (0.000 sec)
```

4. **MIN**

- Purpose: Finds the minimum value in a numeric column or the earliest date in a date column.
- **Syntax:** MIN(column_name)

```
MariaDB [example_database]> SELECT min(obtained_marks) FROM example;
+---------------------+
| min(obtained_marks) |
+---------------------+
|               35.00 |
+---------------------+
1 row in set (0.003 sec)
```

5. MAX

- Purpose: Finds the maximum value in a numeric column or the latest date in a date column.
- **Syntax: MAX(column_name)**

```
MariaDB [example_database]> SELECT max(obtained_marks) FROM example;
+---------------------+
| max(obtained_marks) |
+---------------------+
|               98.00 |
+---------------------+
1 row in set (0.001 sec)
```

5. Grouping Data

GROUP BY

The GROUP BY clause is used to arrange identical data into groups. This is especially useful when you want to perform aggregation operations, such as counting or averaging, on subsets of data.

```
SELECT column1, COUNT(*)
FROM table_name
GROUP BY column1;
```

```
MariaDB [example_database]> select obtained_marks, count(*)
    -> from example
    -> group by obtained_marks;
+----------------+----------+
| obtained_marks | count(*) |
+----------------+----------+
|          35.00 |        1 |
|          50.00 |        1 |
|          56.00 |        1 |
|          67.00 |        1 |
|          77.00 |        1 |
|          90.00 |        1 |
|          98.00 |        1 |
+----------------+----------+
7 rows in set (0.038 sec)
```

HAVING

The HAVING clause is used to filter groups based on a condition, similar to how the WHERE clause filters rows. However, HAVING works with aggregated data that results from the GROUP BY clause.

```
SELECT column1, COUNT(*)
FROM table_name
GROUP BY column1
HAVING COUNT(*) > 5;
```

```
MariaDB [example_database]> select First_name,last_name,obtained_marks, count(*)
    -> from example
    -> group by obtained_marks
    -> having obtained_marks >75;
+------------+-----------+----------------+----------+
| First_name | last_name | obtained_marks | count(*) |
+------------+-----------+----------------+----------+
| sangeetha  | m         |          77.00 |        1 |
| rani       | s         |          90.00 |        1 |
| raju       | p         |          98.00 |        1 |
+------------+-----------+----------------+----------+
3 rows in set (0.039 sec)
```

Important Points:

- The GROUP BY clause must be used with aggregate functions like COUNT, SUM, AVG, MIN, and MAX.
- The HAVING clause is used to apply conditions to the groups created by GROUP BY, while WHERE is used to filter rows before grouping.

Both GROUP BY and HAVING help in summarizing and analysing data effectively, especially when dealing with large datasets and needing insights from aggregate data.

7. Subqueries

A subquery in the SELECT : clause is used to compute a value that can be included in the results of the main query. This is useful when you want to add additional information derived from another query.

```
SELECT column1, (SELECT AVG(salary) FROM employees) AS avg_salary
FROM employees;
```

```
MariaDB [example_database]> SELECT First_name,last_name, (SELECT AVG(obtained_marks) FROM example) AS avg_marks
    -> from example;
+------------+-----------+-----------+
| First_name | last_name | avg_marks |
+------------+-----------+-----------+
| raju       | p         | 67.571429 |
| rani       | s         | 67.571429 |
| rajesh     | s         | 67.571429 |
| sangeetha  | m         | 67.571429 |
| jhanavi    | e         | 67.571429 |
| kiran      | r         | 67.571429 |
| sahana     | r         | 67.571429 |
+------------+-----------+-----------+
7 rows in set (0.001 sec)
```

Subquery in WHERE: Filter results based on a subquery.

A subquery in the WHERE clause is used to filter the results based on the result of another query. It helps in cases where you need to match rows with values computed from another table or query.

```
SELECT column1
FROM table_name
WHERE column2 = (SELECT MAX(column2) FROM table_name);
```

```
MariaDB [example_database]> SELECT First_name,last_name,obtained_marks
    -> from example
    -> where obtained_marks = (select max(obtained_marks) from example);
+------------+-----------+----------------+
| First_name | last_name | obtained_marks |
+------------+-----------+----------------+
| raju       | p         |          98.00 |
+------------+-----------+----------------+
1 row in set (0.001 sec)
```

What Are Indexes?

Indexes are used to speed up the retrieval of rows from a database table. They work similarly to the index in a book, which helps you quickly locate information without having to read the entire book.

Creating Indexes: Improve query performance.

```
CREATE INDEX index_name
ON Table_name (column_name);
```

```
MariaDB [example_database]> create index index_name
    -> on example (last_name);
Query OK, 0 rows affected (0.032 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Types of Indexes

1. **Single-Column Index**: An index on a single column. This is what you've described in your example.

   ```
   CREATE INDEX idx_last_name
   ON Table_name (last_name);
   ```

2. **Composite (Multi-Column) Index**: An index on multiple columns. Useful when queries involve multiple columns in the WHERE clause.

   ```
   CREATE INDEX idx_dept_salary
   ON Table_name (department_id, salary);
   ```

   This index helps with queries that filter by both department_id and salary.

3. **Unique Index**: Ensures that the values in the indexed column(s) are unique across the table.

   ```
   CREATE UNIQUE INDEX idx_email
   ON Table_name (email);
   ```

   This index ensures no two rows can have the same email address.

4. **Full-Text Index**: Used for full-text searches, typically on textual data.

   ```
   CREATE FULLTEXT INDEX idx_description
   ON products (description);
   ```

5. **Spatial Index**: Used for spatial data types, often in GIS applications.

6. Joins

INNER JOIN: An INNER JOIN combines rows from two tables where there is a match between the specified columns. Only rows with matching values in both tables are included in the result set.

```
SELECT table1.column1, table2.column2
FROM table1
INNER JOIN table2 ON table1.common_column = table2.common_column;
```

```
MariaDB [example_database]> SELECT example.first_name, example.last_name, example_2.employee_id, example_2.joining_date
    -> FROM example
    -> INNER JOIN example_2 ON example.obtained_marks = example_2.obtained_marks;
+------------+-----------+-------------+--------------+
| first_name | last_name | employee_id | joining_date |
+------------+-----------+-------------+--------------+
| raju       | p         |          11 | 2024-09-24   |
| rani       | s         |          12 | 2024-09-25   |
| rajesh     | s         |          13 | 2024-09-25   |
| sangeetha  | m         |          14 | 2024-09-26   |
| jhanavi    | e         |          15 | 2024-09-26   |
| kiran      | r         |          16 | 2024-09-26   |
| sahana     | r         |          17 | 2024-09-25   |
| rani       | s         |          19 | 2024-09-25   |
+------------+-----------+-------------+--------------+
8 rows in set (0.001 sec)
```

LEFT JOIN (LEFT OUTER JOIN): A LEFT JOIN (or LEFT OUTER JOIN) returns all rows from the left table and the matched rows from the right table. If there's no match, the result is NULL for columns from the right table.

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

```
MariaDB [example_database]> SELECT example.first_name, example.last_name, example_2.employee_id, example_2.joining_date
    -> FROM example
    -> left JOIN example_2 ON example.obtained_marks = example_2.obtained_marks;
+------------+-----------+-------------+--------------+
| first_name | last_name | employee_id | joining_date |
+------------+-----------+-------------+--------------+
| raju       | p         |          11 | 2024-09-24   |
| rani       | s         |          12 | 2024-09-25   |
| rajesh     | s         |          13 | 2024-09-25   |
| sangeetha  | m         |          14 | 2024-09-26   |
| jhanavi    | e         |          15 | 2024-09-26   |
| kiran      | r         |          16 | 2024-09-26   |
| sahana     | r         |          17 | 2024-09-25   |
| rani       | s         |          19 | 2024-09-25   |
+------------+-----------+-------------+--------------+
8 rows in set (0.001 sec)
```

RIGHT JOIN (RIGHT OUTER JOIN): A RIGHT JOIN (or RIGHT OUTER JOIN) returns all rows from the right table and the matched rows from the left table. If there's no match, the result is NULL for columns from the left table.

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

```
MariaDB [example_database]> SELECT example.first_name, example.last_name, example_2.employee_id, example_2.joining_date
    -> FROM example
    -> RIGHT JOIN example_2 ON example.obtained_marks = example_2.obtained_marks;
+------------+-----------+-------------+--------------+
| first_name | last_name | employee_id | joining_date |
+------------+-----------+-------------+--------------+
| raju       | p         |          11 | 2024-09-24   |
| rani       | s         |          12 | 2024-09-25   |
| rani       | s         |          19 | 2024-09-25   |
| rajesh     | s         |          13 | 2024-09-25   |
| sangeetha  | m         |          14 | 2024-09-26   |
| jhanavi    | e         |          15 | 2024-09-26   |
| kiran      | r         |          16 | 2024-09-26   |
| sahana     | r         |          17 | 2024-09-25   |
| NULL       | NULL      |          18 | 2024-09-25   |
+------------+-----------+-------------+--------------+
9 rows in set (0.001 sec)
```

FULL JOIN (FULL OUTER JOIN): A FULL JOIN (or FULL OUTER JOIN) returns all rows when there is a match in either the left or right table. If there's no match, the result is NULL for columns from the table that does not have a match.

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

CROSS JOIN: A CROSS JOIN returns the Cartesian product of the two tables. It pairs every row of the first table with every row of the second table, resulting in a large result set.

```
SELECT table1.column1, table2.column2
FROM table1
CROSS JOIN table2;
```

```
MariaDB [example_database]> select example.First_name,example_2.employee_id
    -> from example
    -> cross join example_2;
+------------+-------------+
| First_name | employee_id |
+------------+-------------+
| raju       |          11 |
| rani       |          11 |
| rajesh     |          11 |
| sangeetha  |          11 |
| jhanavi    |          11 |
| kiran      |          11 |
| sahana     |          11 |
| raju       |          12 |
| rani       |          12 |
| rajesh     |          12 |
| sangeetha  |          12 |
| jhanavi    |          12 |
| kiran      |          12 |
| sahana     |          12 |
| raju       |          13 |
| rani       |          13 |
| rajesh     |          13 |
| sangeetha  |          13 |
| jhanavi    |          13 |
| kiran      |          13 |
| sahana     |          13 |
| raju       |          14 |
| rani       |          14 |
| rajesh     |          14 |
| sangeetha  |          14 |
```

```
| rani       |          18 |
| rajesh     |          18 |
| sangeetha  |          18 |
| jhanavi    |          18 |
| kiran      |          18 |
| sahana     |          18 |
| raju       |          19 |
| rani       |          19 |
| rajesh     |          19 |
| sangeetha  |          19 |
| jhanavi    |          19 |
| kiran      |          19 |
| sahana     |          19 |
+------------+-------------+
63 rows in set (0.001 sec)
```

Summary of Join Types

- **INNER JOIN**: Returns only matching rows from both tables.
- **LEFT JOIN (LEFT OUTER JOIN)**: Returns all rows from the left table and matching rows from the right table.
- **RIGHT JOIN (RIGHT OUTER JOIN)**: Returns all rows from the right table and matching rows from the left table.
- **FULL JOIN (FULL OUTER JOIN)**: Returns all rows when there is a match in either table.
- **CROSS JOIN**: Returns the Cartesian product of the two tables.

When to Use Each Join Type

- Use **INNER JOIN** when you need rows that have matching values in both tables.
- Use **LEFT JOIN** when you need all rows from the left table and matched rows from the right table.
- Use **RIGHT JOIN** when you need all rows from the right table and matched rows from the left table.
- Use **FULL JOIN** when you need all rows from both tables, regardless of whether there is a match.
- Use **CROSS JOIN** for generating combinations of rows from two tables, though it can produce very large result sets.

The SQL UNION Operator

The UNION operator is used to combine the result-set of two or more 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

SELECT *column_name(s)* FROM *table1*
UNION
SELECT *column_name(s)* FROM *table2*;

```
MariaDB [example_database]> SELECT First_name AS Name, obtained_marks AS Marks
    -> FROM example
    -> UNION
    -> SELECT last_name AS Name, obtained_marks AS Marks
    -> FROM example_2;
+-----------+-------+
| Name      | Marks |
+-----------+-------+
| raju      | 98.00 |
| rani      | 90.00 |
| rajesh    | 67.00 |
| sangeetha | 77.00 |
| jhanavi   | 56.00 |
| kiran     | 50.00 |
| sahana    | 35.00 |
| p         | 98.00 |
| s         | 90.00 |
| s         | 67.00 |
| m         | 77.00 |
| e         | 56.00 |
| r         | 50.00 |
| r         | 35.00 |
| o         | 85.00 |
| a         | 90.00 |
+-----------+-------+
16 rows in set (0.001 sec)
```

SQL UNION ALL

UNION ALL is similar to UNION but with a key difference: UNION ALL does not remove duplicate rows from the result set. It combines all rows from the involved SELECT statements, including duplicates, into a single result set.

```
SELECT column1, column2, ...
FROM table1
UNION ALL
SELECT column1, column2, ...
FROM table2;
```

```
MariaDB [example_database]> SELECT First_name AS name, last_name AS surname, obtained_marks AS details
    -> FROM example
    -> UNION ALL
    -> SELECT employee_id AS name, NULL AS surname, joining_date AS details
    -> FROM example_2;
+-----------+---------+------------+
| name      | surname | details    |
+-----------+---------+------------+
| raju      | p       | 98.00      |
| rani      | s       | 90.00      |
| rajesh    | s       | 67.00      |
| sangeetha | m       | 77.00      |
| jhanavi   | e       | 56.00      |
| kiran     | r       | 50.00      |
| sahana    | r       | 35.00      |
| 11        | NULL    | 2024-09-24 |
| 12        | NULL    | 2024-09-25 |
| 13        | NULL    | 2024-09-25 |
| 14        | NULL    | 2024-09-26 |
| 15        | NULL    | 2024-09-26 |
| 16        | NULL    | 2024-09-26 |
| 17        | NULL    | 2024-09-25 |
| 18        | NULL    | 2024-09-25 |
| 19        | NULL    | 2024-09-25 |
+-----------+---------+------------+
16 rows in set (0.002 sec)
```

Key Points

1. **Duplicates**: Unlike UNION, which removes duplicate rows, UNION ALL includes all duplicates. This can be useful when you need to retain all rows and you don't want the overhead of removing duplicates.
2. **Performance**: UNION ALL is generally faster than UNION because it doesn't need to perform the duplicate elimination process. If you know that the result sets don't contain duplicates or if you want to include duplicates, UNION ALL is often more efficient.
3. **Column Count and Data Types**: Similar to UNION, all SELECT statements involved must have the same number of columns and compatible data types.

11. Views

SQL is a powerful way to simplify complex queries and present data in a more understandable format. A view is essentially a virtual table that is defined by a query. When you query a view, it executes the underlying SQL query and presents the results as if they were a table.

Here's a breakdown of how to create a view:

Syntax

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Components

1. **CREATE VIEW view_name AS**: This command creates a new view with the name view_name.
2. **SELECT column1, column2, ...**: The columns you want to include in the view.
3. **FROM table_name**: The base table or tables from which the data is selected.
4. **WHERE condition**: Conditions to filter the data, similar to the WHERE clause in a standard SELECT statement.

SQL Aliases

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.

Aliases are specified using the AS keyword, but the AS keyword is optional in many SQL implementations. Here's a quick overview of how to use aliases for both tables and columns.

Column Aliases

Column aliases rename columns in the output of your query. This is particularly useful when you want to give more meaningful names to the result set columns or when you use expressions in your query.

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

```
MariaDB [example_database]> select First_name AS first_name
    -> from example;
+------------+
| first_name |
+------------+
| raju       |
| rani       |
| rajesh     |
| sangeetha  |
| jhanavi    |
| kiran      |
| sahana     |
+------------+
7 rows in set (0.001 sec)
```

Table Aliases

Table aliases are used to give a table a temporary name, which can be useful especially when dealing with multiple tables in a query or when using self-joins.

SELECT column_name
FROM table_name AS alias_name;

```
MariaDB [example_database]> select first_name
    -> from example AS example_1;
+------------+
| first_name |
+------------+
| raju       |
| rani       |
| rajesh     |
| sangeetha  |
| jhanavi    |
| kiran      |
| sahana     |
+------------+
7 rows in set (0.001 sec)
```

1. UPDATE

The UPDATE statement modifies existing records in a table.

UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;

```
MariaDB [example_database]> update example
    -> set First_name='rajeshh',obtained_marks=100
    -> where First_name='raju';
Query OK, 1 row affected (0.041 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [example_database]> select *from example
    -> ;
+------------+-----------+----------------+
| First_name | last_name | obtained_marks |
+------------+-----------+----------------+
| rajeshh    | p         |         100.00 |
| rani       | s         |          90.00 |
| rajesh     | s         |          67.00 |
| sangeetha  | m         |          77.00 |
| jhanavi    | e         |          56.00 |
| kiran      | r         |          50.00 |
| sahana     | r         |          35.00 |
+------------+-----------+----------------+
7 rows in set (0.000 sec)
```

2. DELETE

The DELETE statement removes records from a table.

```
DELETE FROM table_name
WHERE condition;
```

```
MariaDB [example_database]> delete from example
    -> where obtained_marks=35;
Query OK, 0 rows affected (0.000 sec)

MariaDB [example_database]> select* from example;
+------------+-----------+-----------------+
| First_name | last_name | obtained_marks  |
+------------+-----------+-----------------+
| rajeshh    | p         |          100.00 |
| rani       | s         |           90.00 |
| rajesh     | s         |           67.00 |
| sangeetha  | m         |           77.00 |
| jhanavi    | e         |           56.00 |
| kiran      | r         |           50.00 |
+------------+-----------+-----------------+
6 rows in set (0.001 sec)
```

5. ALTER DATABASE

The ALTER DATABASE statement modifies an existing database.

```
ALTER DATABASE database_name
MODIFY DATABASE_OPTIONS;
```

7. ALTER TABLE

The ALTER TABLE statement modifies an existing table, such as adding, deleting, or changing columns.

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

8. DROP TABLE

The DROP TABLE statement deletes a table and all its data.

```
DROP TABLE table_name;
```

9. CREATE INDEX

The CREATE INDEX statement creates an index (search key) on a table to improve query performance.

```
CREATE INDEX index_name
ON table_name (column_name);
```

10. DROP INDEX

The DROP INDEX statement removes an index from a table.

```
DROP INDEX index_name
ON table_name;
```

Exit and Quit commands

In SQL, particularly when using command-line interfaces or database management tools, EXIT and QUIT are commands used to terminate the current session or exit from the SQL command-line interface. While these commands serve a similar purpose, their usage might vary depending on the database management system (DBMS) and the environment in which you're working.

Common Usage

1. **EXIT Command**
    - **Purpose**: The EXIT command is used to exit from the SQL command-line interface or client.
    - **Usage**: Typically used in many SQL command-line tools like MySQL, MariaDB, and PostgreSQL.
      When you enter EXIT; in a SQL command-line tool, it will close the session and return you to the system shell or command prompt.
2. **QUIT Command**
    - **Purpose**: The QUIT command also exits from the SQL command-line interface or client.
    - **Usage**: Commonly used in MySQL and MariaDB command-line tools.

   Similar to EXIT, when you use QUIT; it closes the SQL command-line session and returns you to the command prompt.

Differences and Similarities

- **Interchangeability**: In many SQL clients, EXIT and QUIT can be used interchangeably to end the session. They effectively perform the same action.
- **Environment Specifics**: The availability of EXIT or QUIT can depend on the SQL client or environment you are using. For instance, some SQL environments or tools might recognize one but not the other.

Summary

- **EXIT**: Commonly used to terminate a SQL command-line session in various SQL clients.
- **QUIT**: Also used to end a SQL command-line session, similar to EXIT.

Both commands are generally used to close your SQL command-line interface and return to your operating system’s command prompt or shell.