**PART-1**

**\*\*DBMS\*\* (Database Management System)**

**1. In DBMS, data are stored as a file.**

**2. Normalization is not available in DBMS.**

**3. It allows one user at a time.**

**4. Hierarchical arrangement of data.**

**5. It does not support ACID properties.**

**6. It does not support client-server architecture.**

**\*\*RDBMS\*\* (Relational Database Management System)**

**1. In RDBMS, data are stored in a tabular form.**

**2. Normalization is available in RDBMS.**

**3. It allows more than one user at a time.**

**4. Stores data in the form of rows and columns.**

**5. It supports ACID properties.**

**6. It supports client-server architecture.**

**ACID properties are a set of key principles that ensure reliable processing of database transactions, particularly in relational databases like RDBMS. The acronym ACID stands for:**

**1. \*\*Atomicity\*\*:**

**- A transaction is considered atomic if either all of its operations are completed successfully, or none of them are applied. This ensures that no partial updates are made if a transaction fails. It treats transactions as an "all-or-nothing" unit.**

**2. \*\*Consistency\*\*:**

**- A transaction must transition the database from one valid state to another, ensuring that data remains valid according to defined rules (constraints, triggers, etc.). After the transaction is completed, the database should be consistent.­­­­**

**3. \*\*Isolation\*\*:**

**- Transactions are executed independently of one another, even when running concurrently. This means that the execution of a transaction is isolated from others, preventing issues such as dirty reads or conflicting writes. Isolation ensures that intermediate transaction states are not visible to other transactions.**

**4. \*\*Durability\*\*:**

**- Once a transaction is successfully committed, it is permanently applied to the database, even in the case of system crashes or failures. The changes made by the transaction are stored in a way that they can be recovered after a failure.**

**These properties are essential to ensure the reliability, correctness, and robustness of database operations.**

1. Difference Between Relational and Non-relational Databases

Relational Database:

- Definition: A relational database stores data in predefined schemas, using tables with rows and columns. It ensures data consistency, integrity, and relationships using primary and foreign keys.

- Example: MySQL, PostgreSQL, Microsoft SQL Server.

- Use Case: Best for structured data, such as customer information in an e-commerce website.

Example Query (SQL):

SELECT \* FROM Customers WHERE CustomerID = 1;

Non-relational Database:

- Definition: Non-relational databases (NoSQL) store data in formats like documents, key-value pairs, graphs, or wide columns without a fixed schema.

- Example: MongoDB, Cassandra, DynamoDB.

- Use Case: Suitable for handling large amounts of unstructured or semi-structured data, such as user activity logs or social media data.

Example Query (MongoDB):

db.customers.find({ "CustomerID": 1 });

2. SQL Keywords and Commands (Case Insensitivity)

- SQL keywords are \*\*NOT case-sensitive\*\*, which means SELECT and select are treated the same.

Common SQL Commands:

- SELECT: Extracts data from a database.

- UPDATE: Updates data in a database. **[DML]**

- 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. **[DDL]**

- 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.

3. SELECT DISTINCT Statement

- Purpose: Returns only distinct (non-duplicate) values from a column.

Example Query:

SELECT DISTINCT Country FROM Customers;

Count Distinct Values Example:

SELECT COUNT(DISTINCT Country) FROM Customers;

4. Operators in SQL Database

- Operators are used in SQL to perform operations on data. Common operators include:

- Arithmetic Operators: +, -, \*, /, %

- Comparison Operators: =, >, <, >=, <=, <> (not equal)

- Logical Operators: AND, OR, NOT

5. Data Types in SQL and Their Ranges

Common Data Types:

- INT: Stores whole numbers. Range: -2147483648 to 2147483647.

- VARCHAR(n): Stores variable-length text. Max length is defined by n.

- DECIMAL(p,s): Stores fixed-point numbers, where p is the precision and s is the scale.

- DATE: Stores date values in YYYY-MM-DD format.

6. MySQL ORDER BY Keyword

- Purpose: Sorts the result set in ascending (ASC) or descending (DESC) order.

Example Query:

SELECT \* FROM Customers ORDER BY Country ASC, CustomerName DESC;

7. MySQL INSERT INTO Statement

-Syntax:

1. Specify columns and values:

INSERT INTO table\_name (column1, column2) VALUES (value1, value2);

2. Insert values into all columns:

INSERT INTO table\_name VALUES (value1, value2, value3);

8. NULL Value in SQL

- NULL represents a field with no value. It is different from zero or empty spaces.

- Testing for NULL:

SELECT column\_names FROM table\_name WHERE column\_name IS NULL;

9. MySQL UPDATE Statement

- Purpose: Modifies existing records in a table.

Example Query:

UPDATE Customers SET ContactName = 'John Doe' WHERE CustomerID = 1;

10. MySQL DELETE Statement

- Purpose: Deletes records from a table.

Example Query:

DELETE FROM Customers WHERE CustomerID = 1;

11. MySQL LIMIT Clause

- Purpose: Specifies the number of records to return from a query.

Example Query (with OFFSET):

SELECT \* FROM Customers LIMIT 3 OFFSET 3;

12. MySQL MIN() and MAX() Functions

- MIN(): Returns the smallest value.

- MAX(): Returns the largest value.

Example:

SELECT MIN(Price) AS SmallestPrice FROM Products;

SELECT MAX(Price) AS LargestPrice FROM Products;

13. MySQL COUNT(), AVG(), SUM() Functions

- COUNT(): Returns the number of rows that match a condition.

- AVG(): Returns the average value.

- SUM(): Returns the total sum of a numeric column.

Example:

SELECT COUNT(CustomerID) FROM Customers WHERE Country = 'Germany';

SELECT AVG(Salary) FROM Employees;

SELECT SUM(Salary) FROM Employees;

14. MySQL LIKE Operator

- Purpose: Searches for a specified pattern in a column.

Example Query:

SELECT \* FROM Customers WHERE CustomerName LIKE 'A%';

LIKE Operator Description

WHERE CustomerName LIKE 'a%' Finds any values that start with "a"

WHERE CustomerName LIKE '%a' Finds any values that end with "a"

WHERE CustomerName LIKE '%or%' Finds any values that have "or" in any position

WHERE CustomerName LIKE '\_r%' Finds any values that have "r" in the second position

WHERE CustomerName LIKE 'a\_%' Finds any values that start with "a" and are at least 2 characters in length

WHERE CustomerName LIKE 'a\_\_%' Finds any values that start with "a" and are at least 3 characters in length

WHERE ContactName LIKE 'a%o' Finds any values that start with "a" and ends with "o"

Key Takeaways for Interview:

- Relational vs. Non-relational databases: Emphasize structured schema and use cases.

- SQL Queries: Focus on common operations (SELECT, INSERT, UPDATE, DELETE), data manipulation, and performance optimization (LIMIT, DISTINCT).

- NULL and Testing for NULL: Highlight how NULL differs from other values and the correct way to handle it in queries.

**1.SQL LIKE Operator & Wildcards**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. You can use wildcard characters to match patterns.

Wildcard Characters:

- %: Represents zero or more characters.

- \_: Represents a single character.

Example: Selecting Names Starting with "a"

SELECT \* FROM Customers

WHERE CustomerName LIKE 'a%';

This query selects all customers whose names start with the letter "a".

Example: Selecting Cities Starting with "ber"

SELECT \* FROM Customers

WHERE City LIKE 'ber%';

This query selects all customers with a city name starting with "ber" (e.g., Berlin, Bern).

Example: Single Character Wildcard

SELECT \* FROM Customers

WHERE City LIKE '\_ondon';

This query selects all cities where the second character onward matches "ondon" (e.g., London).

**2. SQL IN Operator**

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

Example: Using IN

SELECT \* FROM Customers

WHERE Country IN ('Germany', 'France', 'UK');

This selects all customers located in Germany, France, or the UK.

Example: Using NOT IN

SELECT \* FROM Customers

WHERE Country NOT IN ('Germany', 'France', 'UK');

This selects all customers not located in Germany, France, or the UK.

**3. SQL BETWEEN Operator**

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates, and the range is inclusive of both the start and end values.

Example: BETWEEN Numbers

SELECT \* FROM Products

WHERE Price BETWEEN 10 AND 20;

This query selects products priced between 10 and 20.

Example: NOT BETWEEN

SELECT \* FROM Products

WHERE Price NOT BETWEEN 10 AND 20;

This selects products priced outside the range of 10 to 20.

Example: BETWEEN with IN

SELECT \* FROM Products

WHERE Price BETWEEN 10 AND 20

AND CategoryID NOT IN (1, 2, 3);

This selects products with prices between 10 and 20, but excludes products in categories 1, 2, or 3.

Example: BETWEEN Text Values

SELECT \* FROM Products

WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'

ORDER BY ProductName;

This selects products with names alphabetically between 'Carnarvon Tigers' and 'Mozzarella di Giovanni'.

4. SQL Aliases

Aliases are used to give a table or a column a temporary name, which only exists for the duration of the query. Aliases are useful to make the column names more readable.

Example: Column Alias

SELECT CustomerName, CONCAT\_WS(', ', Address, PostalCode, City, Country) AS Address

FROM Customers;

This query combines multiple columns (Address, PostalCode, City, Country) into one column named "Address".

5. SQL JOIN Types

A JOIN clause is used to combine rows from two or more tables based on a related column.

Types of Joins in MySQL:

- INNER JOIN: Returns records that have matching values in both tables.

- LEFT JOIN: Returns all records from the left table and the matched records from the right table.

- RIGHT JOIN: Returns all records from the right table and the matched records from the left table.

- CROSS JOIN: Returns the Cartesian product of both tables (every combination of rows).

Example: INNER JOIN

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

INNER JOIN Orders

ON Customers.CustomerID = Orders.CustomerID;

This query selects records where there is a match between CustomerID in the Customers and Orders tables.

Example: LEFT JOIN

SELECT Customers.CustomerName, Orders.OrderID

FROM Customers

LEFT JOIN Orders

ON Customers.CustomerID = Orders.CustomerID;

This query returns all customers, along with their corresponding orders, if they exist. If a customer does not have an order, NULL is shown.

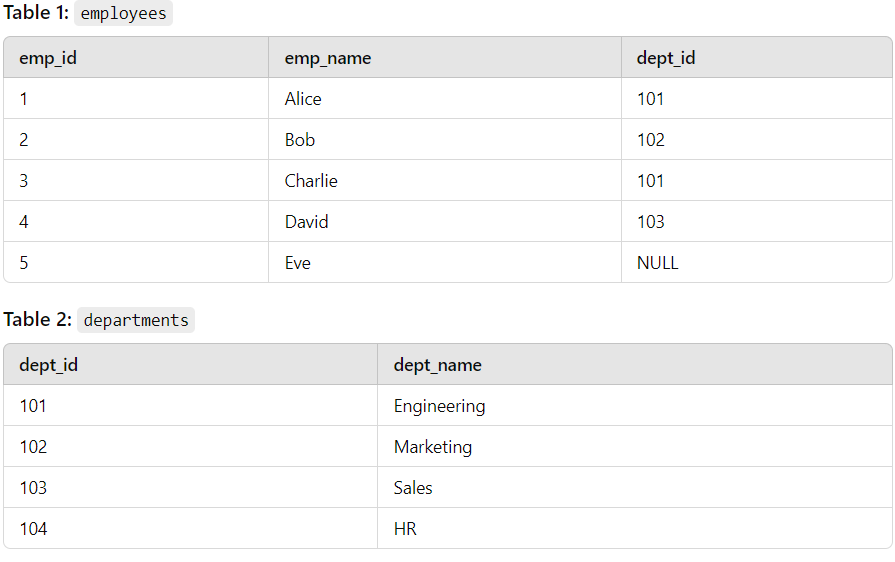
Interview Focus Points:

- LIKE Operator with Wildcards: Explain how % and \_ are used to match patterns in strings, and when to use LIKE instead of = for partial matches.

- IN and BETWEEN: Understand how these operators simplify complex OR and range conditions.

- Joins: Explain the difference between INNER, LEFT, RIGHT, and CROSS joins with real-world examples (e.g., customers and orders).



#### 1. **INNER JOIN**

This will return records that have matching values in both tables.

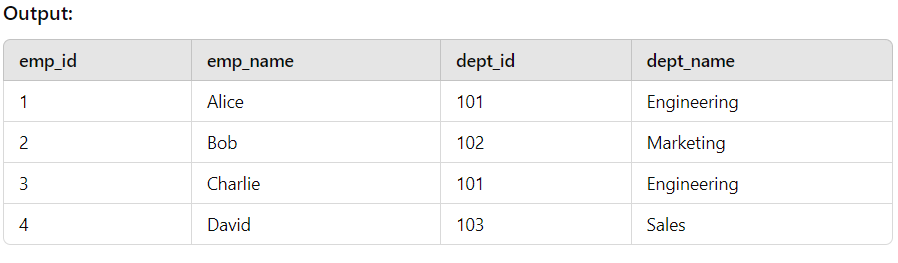
**Query:**

SELECT e.emp\_id, e.emp\_name, e.dept\_id, d.dept\_name

FROM employees e

INNER JOIN departments d

ON e.dept\_id = d.dept\_id;



#### 2. **LEFT JOIN** (or LEFT OUTER JOIN)

Returns all records from the left table (employees), and the matched records from the right table (departments). The result is NULL from the right side if there is no match.

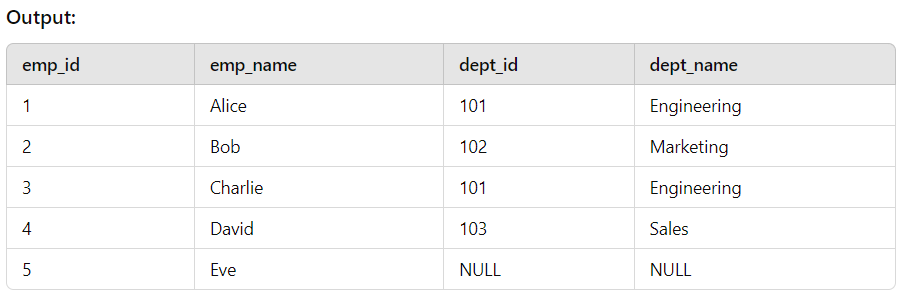
**Query:**

SELECT e.emp\_id, e.emp\_name, e.dept\_id, d.dept\_name

FROM employees e

LEFT JOIN departments d

ON e.dept\_id = d.dept\_id;



#### 3. **RIGHT JOIN** (or RIGHT OUTER JOIN)

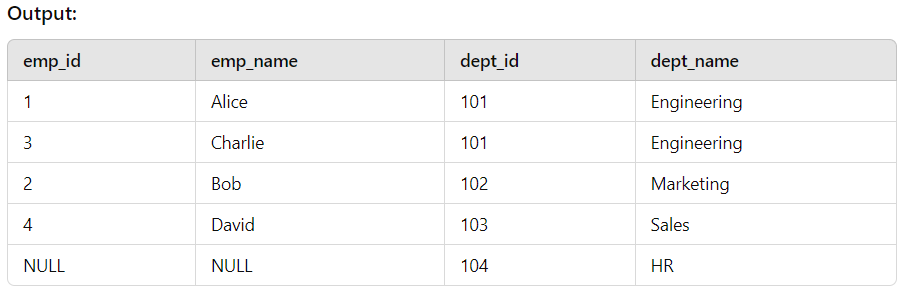
Returns all records from the right table (departments), and the matched records from the left table (employees). The result is NULL from the left side when there is no match.

SELECT e.emp\_id, e.emp\_name, e.dept\_id, d.dept\_name

FROM employees e

RIGHT JOIN departments d

ON e.dept\_id = d.dept\_id;



#### 4. **FULL OUTER JOIN**

Returns all records when there is a match in either left (employees) or right (departments) table. If there is no match, the result is NULL on the side that doesn’t have a match.

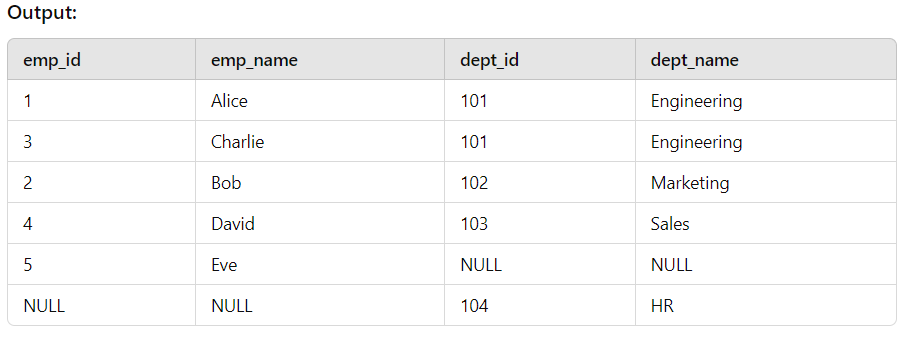
**Query:**

SELECT e.emp\_id, e.emp\_name, e.dept\_id, d.dept\_name

FROM employees e

FULL OUTER JOIN departments d

ON e.dept\_id = d.dept\_id;



CROSS JOIN can potentially return very large result-sets!

NOTES:

A **Cross Origin Join** is not a standard term in SQL. However, based on common terminology, you might be referring to a **Cross Join**. A **Cross Join** in SQL combines every row from two or more tables, resulting in a Cartesian product. It's different from a **FULL OUTER JOIN** and has a distinct purpose.

### 1. **Cross Join**:

* A **Cross Join** returns the Cartesian product of two tables, meaning it matches every row from the first table with every row from the second table.
* It doesn't require any condition to join the tables.

#### Example of Cross Join:

SELECT \*

FROM employees

CROSS JOIN departments;

* If the employees table has 5 rows and the departments table has 3 rows, the result will have 5 × 3 = 15 rows.

### 2. **FULL OUTER JOIN**:

* A **FULL OUTER JOIN** returns all the rows when there is a match in either of the tables, and if there is no match, it returns NULL for the missing data.
* It combines the results of both **LEFT JOIN** and **RIGHT JOIN**.

#### Example of FULL OUTER JOIN:

SELECT \*

FROM employees

FULL OUTER JOIN departments

ON employees.department\_id = departments.id;

* If there are employees without matching departments or departments without matching employees, it will show NULL for the missing data on either side.

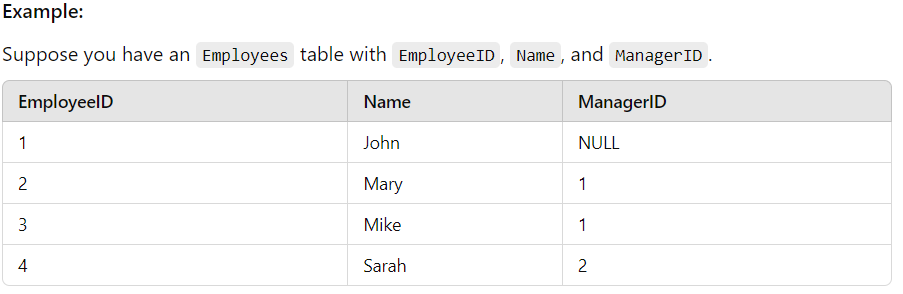
### Key Differences:

* **Cross Join** results in a Cartesian product (every row from one table is combined with every row from the other).
* **FULL OUTER JOIN** returns matched rows from both tables and includes unmatched rows with NULL for missing columns on either side.

**NOTES:**

### 1. **Self Join**:

A **self join** is a join in which a table is joined with itself. It is useful when you want to compare rows within the same table.



To find all employees and their managers:

SELECT e1.Name AS Employee, e2.Name AS Manager

FROM Employees e1

LEFT JOIN Employees e2 ON e1.ManagerID = e2.EmployeeID;

### 2. **UNION**:

The **UNION** operator combines the results of two or more SELECT queries. It returns distinct values by default.

#### Example:

Suppose you have two tables, Students\_2023 and Students\_2024, and you want to get a list of all students from both years without duplicates.

SELECT Name FROM Students\_2023

UNION

SELECT Name FROM Students\_2024;

If you want to include duplicates, use UNION ALL:

SELECT Name FROM Students\_2023

UNION ALL

SELECT Name FROM Students\_2024;

### 3. **GROUP BY**:

The **GROUP BY** statement groups rows that have the same values into summary rows, often used with aggregate functions (COUNT, MAX, MIN, SUM, etc.).

#### Example:

Suppose you have an Orders table, and you want to count the number of orders per customer.

SELECT CustomerID, COUNT(OrderID) AS OrderCount

FROM Orders

GROUP BY CustomerID;

### 4. **HAVING**:

The **HAVING** clause is like WHERE, but it is used to filter data after the GROUP BY operation (usually for aggregate functions).

#### Example:

To find customers who have placed more than 5 orders:

SELECT CustomerID, COUNT(OrderID) AS OrderCount

FROM Orders

GROUP BY CustomerID

HAVING COUNT(OrderID) > 5;

### 5. **EXISTS**:

The **EXISTS** operator is used to test for the existence of any records in a subquery. It returns TRUE if the subquery returns one or more records.

#### Example:

To find all customers who have placed an order:

SELECT CustomerID

FROM Customers c

WHERE EXISTS (SELECT 1 FROM Orders o WHERE o.CustomerID = c.CustomerID);

### 6. **ANY**:

The **ANY** operator compares a value to each value in a list or subquery and returns TRUE if any of the comparisons are true.

#### Example:

To find all customers who have placed an order with an amount greater than any order in Orders2022:

SELECT CustomerID

FROM Orders

WHERE Amount > ANY (SELECT Amount FROM Orders2022);

### 7. **ALL**:

The **ALL** operator compares a value to all values in a list or subquery and returns TRUE only if all comparisons are true.

#### Example:

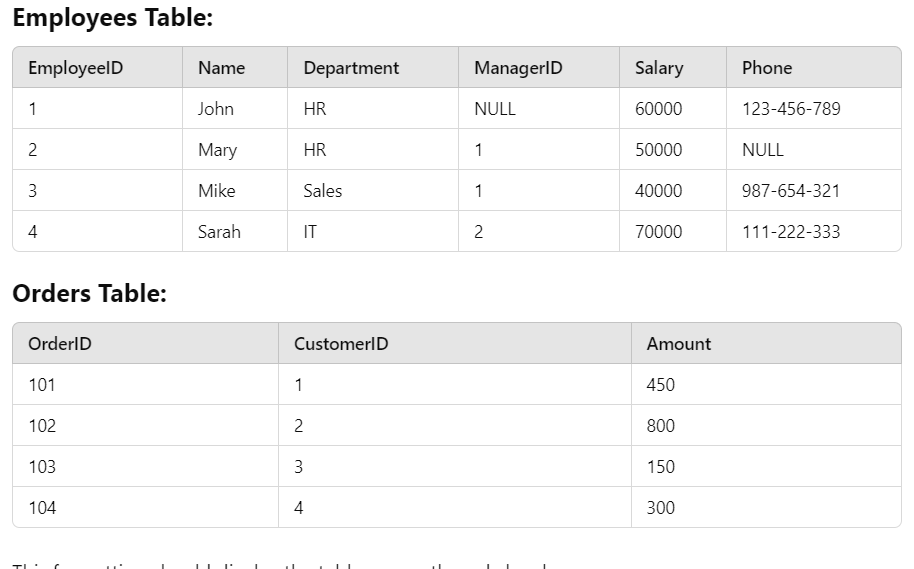
To find customers who have placed an order with an amount greater than all orders in Orders2022:

SELECT CustomerID

FROM Orders

WHERE Amount > ALL (SELECT Amount FROM Orders2022);

**NOTES:**



### **1. INSERT INTO SELECT**

Insert employees from the Employees table who work in the HR department into a new HR\_Employees table.

INSERT INTO HR\_Employees (EmployeeID, Name, Department)

SELECT EmployeeID, Name, Department

FROM Employees

WHERE Department = 'HR';

### **2. CASE**

Categorize employees' salaries as Low, Medium, or High.

SELECT EmployeeID, Name,

CASE

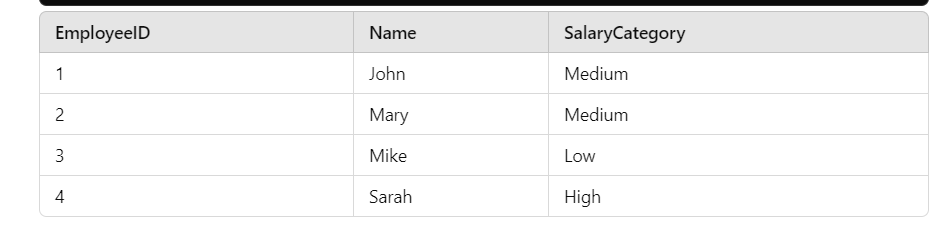
WHEN Salary > 60000 THEN 'High'

WHEN Salary BETWEEN 40000 AND 60000 THEN 'Medium'

ELSE 'Low'

END AS SalaryCategory

FROM Employees;

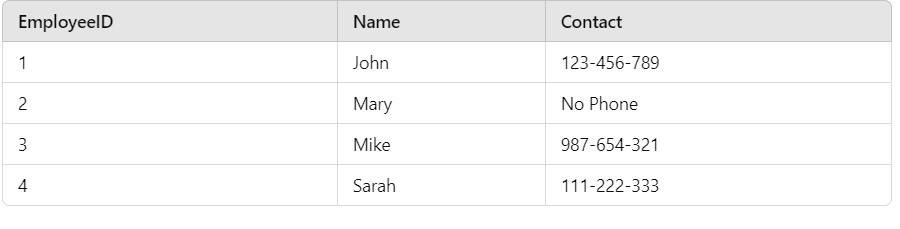


### **3. NULL Functions**

Use **COALESCE** to show the employee’s phone number, and if it’s missing (NULL), show "No Phone".

SELECT EmployeeID, Name, COALESCE(Phone, 'No Phone') AS Contact

FROM Employees;



### **4. COMMENTS**

Adding comments to explain the query logic.

/\* This query retrieves employee names and their respective salary categories \*/

SELECT Name,

CASE

WHEN Salary > 60000 THEN 'High'

ELSE 'Low'

END AS SalaryCategory

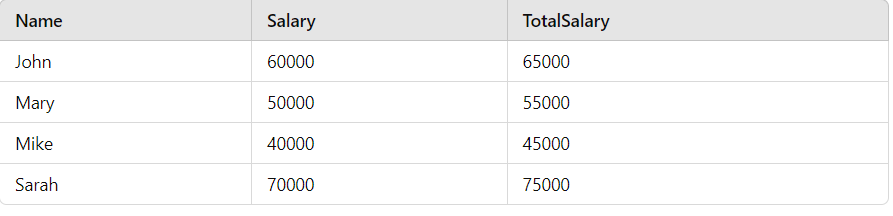
FROM Employees; -- Fetching from the Employees table

### **5. MySQL Operators**

* **Arithmetic Operator**: Adding bonus to salary.

SELECT Name, Salary, Salary + 5000 AS TotalSalary

FROM Employees;

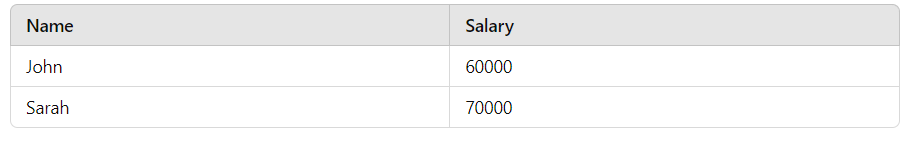


* **Comparison Operator**: Fetch employees with salary greater than 50000.

SELECT Name, Salary

FROM Employees

WHERE Salary > 50000;

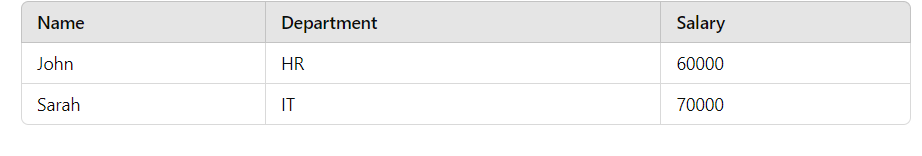


* **Logical Operator**: Employees in either HR or IT with salary > 50000.

SELECT Name, Department, Salary

FROM Employees

WHERE (Department = 'HR' OR Department = 'IT') AND Salary > 50000;



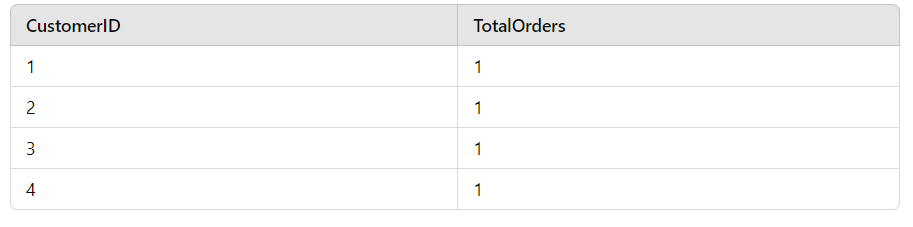
### **6. GROUP BY**

Find the total number of orders placed by each customer.

SELECT CustomerID, COUNT(OrderID) AS TotalOrders

FROM Orders

GROUP BY CustomerID;



### **7. HAVING**

Find customers who have placed more than 1 order (after grouping).

SELECT CustomerID, COUNT(OrderID) AS TotalOrders

FROM Orders

GROUP BY CustomerID

HAVING COUNT(OrderID) > 1;

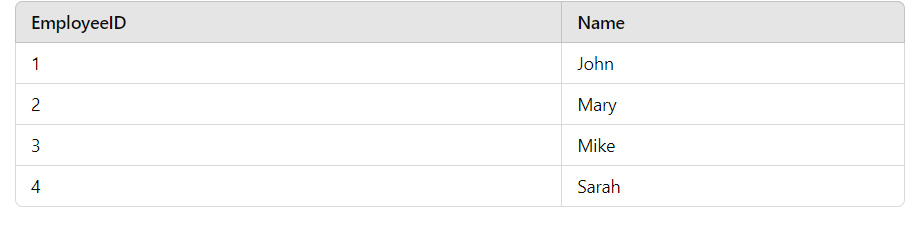
### **8. EXISTS**

Find employees who have placed an order.

SELECT EmployeeID, Name

FROM Employees e

WHERE EXISTS (SELECT 1 FROM Orders o WHERE o.CustomerID = e.EmployeeID);



### **9. ANY**

Find customers who have placed an order with an amount greater than **any** order in Orders2023.

SELECT CustomerID, Amount

FROM Orders

WHERE Amount > ANY (SELECT Amount FROM Orders2023);

### **10. ALL**

Find customers who have placed an order with an amount greater than **all** orders in Orders2023.

SELECT CustomerID, Amount

FROM Orders

WHERE Amount > ALL (SELECT Amount FROM Orders2023);

**PART-2**

SQL Database and Table Operations

1. Creating a Database

The `CREATE DATABASE` statement is used to create a new SQL database. The basic syntax is as follows:

CREATE DATABASE database\_name;

- `database\_name`: This is the name you want to give to your new database. Choose a name that is descriptive and follows the naming conventions of your SQL system.

Example:

CREATE DATABASE SchoolDB;

This command will create a database named `SchoolDB`.

2. Dropping a Database

The `DROP DATABASE` statement is used to delete an existing SQL database. The basic syntax is:

DROP DATABASE database\_name;

- `database\_name`: This is the name of the database you want to delete. Be cautious while using this command, as it will permanently remove the database along with all its data and objects.

Example:

DROP DATABASE SchoolDB;

This command will permanently delete the `SchoolDB` database.

3. Creating a Table

The `CREATE TABLE` statement is used to create a new table within a database. The syntax is:

CREATE TABLE table\_name (

column1 datatype,

column2 datatype,

column3 datatype,

...

);

- `table\_name`: The name of the table you want to create.

- `column1, column2, column3, ...`\*\*: The names of the columns in the table.

- `datatype`: The type of data that the column can store, such as `VARCHAR`, `INTEGER`, `DATE`, etc.

Example:

CREATE TABLE Students (

StudentID INT,

FirstName VARCHAR(50),

LastName VARCHAR(50),

DateOfBirth DATE

);

This command creates a table named `Students` with four columns:

- `StudentID`: Stores integers, likely used as a unique identifier.

- `FirstName`: Stores strings up to 50 characters.

- `LastName`: Stores strings up to 50 characters.

- `DateOfBirth`: Stores date values.

4. Column Parameters and Data Types

- Column Parameters: These specify the names of the columns in the table. Each column represents a specific type of data you want to store, such as student names, IDs, or dates of birth.

- Data Types: These define the type of data each column can hold. Common data types include:

- VARCHAR(size): A variable-length string. `size` defines the maximum number of characters.

- `INT`: An integer.

- `DATE`: A date value.

Key Points to Remember

1. Creating and Dropping Databases: Use `CREATE DATABASE` to create a new database and `DROP DATABASE` to delete an existing one.

2. Creating Tables: Use `CREATE TABLE` to define a new table structure with specified columns and data types.

3. Column Definitions: Each column should have a unique name and an appropriate data type that suits the kind of data you intend to store.

**SQL Table Operations and Constraints**

1. Creating a Table

The `CREATE TABLE` statement is used to create a new table in an SQL database. This table can store data in a structured format, where each column represents a specific type of data.

Syntax:

CREATE TABLE table\_name (

column1 datatype,

column2 datatype,

column3 datatype,

...

);

- `table\_name`: The name of the new table.

- `column1, column2, ...`: Names of the columns to be created in the table.

- `datatype`: The type of data the column can hold, such as `VARCHAR`, `INT`, `DATE`, etc.

Example:

CREATE TABLE Persons (

PersonID INT,

LastName VARCHAR(255),

FirstName VARCHAR(255),

Address VARCHAR(255),

City VARCHAR(255)

);

This command creates a table named `Persons` with the specified columns.

2. Creating a Table Using Another Table

You can create a new table by copying the structure and/or data from an existing table using the `CREATE TABLE AS` statement.

Syntax:

CREATE TABLE new\_table\_name AS

SELECT column1, column2, ...

FROM existing\_table\_name

WHERE ...;

- `new\_table\_name`: The name of the new table.

- `existing\_table\_name`: The name of the table from which to copy data.

Example:

CREATE TABLE TestTable AS

SELECT customername, contactname

FROM customers;

This command creates a new table `TestTable` by copying the `customername` and `contactname` columns from the `customers` table.

3. Dropping a Table

The `DROP TABLE` statement is used to delete an existing table from the database.

Syntax:

DROP TABLE table\_name;

- `table\_name`: The name of the table to be dropped.

Example:

DROP TABLE Persons;

This command deletes the `Persons` table from the database.

4. Truncating a Table

The `TRUNCATE TABLE` statement deletes all data inside a table but keeps the table structure intact.

Syntax:

TRUNCATE TABLE table\_name;

- `table\_name`: The name of the table to be truncated.

Example:

TRUNCATE TABLE Persons;

This command removes all records from the `Persons` table but retains its structure.

5. Altering a Table

The `ALTER TABLE` statement is used to modify an existing table. You can use it to add, delete, or modify columns, and to add or drop constraints.

**Adding a Column**

Syntax:

ALTER TABLE table\_name

ADD column\_name datatype;

Example:

ALTER TABLE Customers

ADD Email VARCHAR(255);

This command adds an `Email` column to the `Customers` table.

**Dropping a Column**

Syntax:

ALTER TABLE table\_name

DROP COLUMN column\_name;

Example:

ALTER TABLE Customers

DROP COLUMN Email;

This command deletes the `Email` column from the `Customers` table.

**Modifying a Column**

Syntax:

ALTER TABLE table\_name

MODIFY COLUMN column\_name datatype;

Example:

ALTER TABLE Persons

MODIFY COLUMN Age INT NOT NULL;

This command changes the data type of the `Age` column and adds a `NOT NULL` constraint.

**6. Constraints in SQL**

Constraints are used to specify rules for the data in a table. They ensure data accuracy and integrity. Constraints can be applied at the column level or the table level.

Common SQL Constraints

1. NOT NULL: Ensures that a column cannot have a NULL value.

2. UNIQUE: Ensures that all values in a column are unique.

3. PRIMARY KEY: A combination of `NOT NULL` and `UNIQUE`. Uniquely identifies each row in a table.

4. FOREIGN KEY: Ensures referential integrity by linking a column to the primary key of another table.

5. CHECK: Ensures that all values in a column satisfy a specific condition.

6. DEFAULT: Provides a default value for a column if none is specified.

7. CREATE INDEX: Used to create an index for faster data retrieval.

Adding Constraints on Table Creation

Syntax:

CREATE TABLE table\_name (

column1 datatype constraint,

column2 datatype constraint,

...

);

Example:

CREATE TABLE Persons (

ID INT NOT NULL,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

Age INT,

CONSTRAINT UC\_Person UNIQUE (ID, LastName)

);

This command creates a table `Persons` with a unique constraint on the combination of the `ID` and `LastName` columns.

Adding Constraints on Existing Tables

Example:

ALTER TABLE Persons

ADD CONSTRAINT UC\_Person UNIQUE (ID, LastName);

This command adds a unique constraint to the `Persons` table for the `ID` and `LastName` columns.

**7. Removing Constraints**

Example:

ALTER TABLE Persons

DROP INDEX UC\_Person;

This command removes the unique constraint named `UC\_Person` from the `Persons` table.

**8. Primary Key Constraint**

The `PRIMARY KEY` uniquely identifies each record in a table. It combines the features of `NOT NULL` and `UNIQUE` constraints.

Example:

CREATE TABLE Persons (

ID INT NOT NULL,

LastName VARCHAR(255) NOT NULL,

FirstName VARCHAR(255),

Age INT,

PRIMARY KEY (ID)

);

This command creates a primary key on the `ID` column of the `Persons` table.

**9. Foreign Key Constraint**

A `FOREIGN KEY` is used to link two tables together. It ensures that the values in a column match values in the primary key column of another table.

Example:

CREATE TABLE Orders (

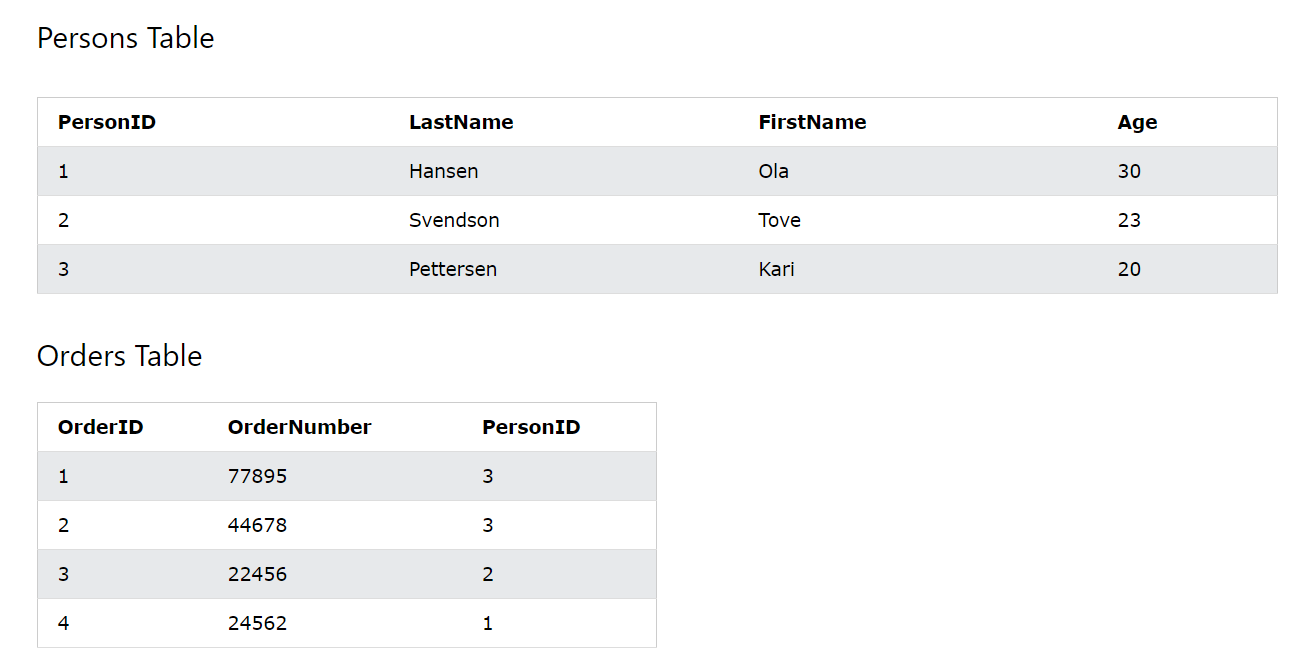
OrderID INT PRIMARY KEY,

CustomerID INT,

FOREIGN KEY (CustomerID) REFERENCES Customers(ID)

);

This command creates a foreign key on the `CustomerID` column of the `Orders` table, referencing the `ID` column of the `Customers` table.



**Foreign Key Constraint in MySQL**

- A foreign key is a field (or collection of fields) in one table that uniquely identifies a row of another table.

- This relationship is established between two tables to enforce referential integrity.

- The foreign key in the child table references the primary key in the parent table.

Explanation:

- In the example below, the `PersonID` column in the `Orders` table points to the `PersonID` column in the `Persons` table.

- The `PersonID` column in the `Persons` table is the primary key, ensuring that each `PersonID` is unique.

- The `PersonID` column in the `Orders` table is a foreign key, ensuring that each value must exist in the `Persons` table.

Example:

Table 1: Persons

| PersonID | LastName | FirstName | Age |

|----------|----------|-----------|-----|

| 1 | Smith | John | 30 |

| 2 | Johnson | Jane | 25 |

| 3 | Williams | Emily | 22 |

Table 2: Orders

| OrderID | OrderNumber | PersonID |

|---------|-------------|----------|

| 101 | 1001 | 1 |

| 102 | 1002 | 2 |

| 103 | 1003 | 3 |

In the `Orders` table, the `PersonID` column references the `PersonID` column in the `Persons` table, establishing a relationship between the orders and the people who made them.

**Creating a Foreign Key on Table Creation**

CREATE TABLE Orders (

OrderID int NOT NULL,

OrderNumber int NOT NULL,

PersonID int,

PRIMARY KEY (OrderID),

FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)

);

This SQL command creates a foreign key on the `PersonID` column in the `Orders` table that references the `PersonID` column in the `Persons` table.

**Naming a Foreign Key Constraint**

CREATE TABLE Orders (

OrderID int NOT NULL,

OrderNumber int NOT NULL,

PersonID int,

PRIMARY KEY (OrderID),

CONSTRAINT FK\_PersonOrder FOREIGN KEY (PersonID)

REFERENCES Persons(PersonID)

);

This example names the foreign key constraint as `FK\_PersonOrder`.

Altering an Existing Table to Add a Foreign Key

**SQL Command:**

**ALTER TABLE Orders**

ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

This command adds a foreign key to the `Orders` table after it has been created.

Naming a Foreign Key Constraint on an Existing Table

ALTER TABLE Orders

ADD CONSTRAINT FK\_PersonOrder

FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

This command adds a named foreign key constraint `FK\_PersonOrder` to the `Orders` table.

Dropping a Foreign Key Constraint

SQL Command:

ALTER TABLE Orders

DROP FOREIGN KEY FK\_PersonOrder;

This command drops the foreign key constraint `FK\_PersonOrder` from the `Orders` table.

Check Constraint in MySQL

Concept:

- The CHECK constraint is used to limit the value range that can be placed in a column.

- It ensures that the data in the column meets specific criteria.

Example:

Table: Persons

| ID | LastName | FirstName | Age | City |

|----|----------|-----------|-----|----------|

| 1 | Smith | John | 30 | Sandnes |

| 2 | Johnson | Jane | 25 | Oslo |

| 3 | Williams | Emily | 22 | Bergen |

Creating a CHECK Constraint on Table Creation

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

CHECK (Age >= 18)

);

This ensures that the `Age` column only allows values 18 or older.

Naming a CHECK Constraint

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

City varchar(255),

CONSTRAINT CHK\_Person CHECK (Age >= 18 AND City = 'Sandnes')

);

This constraint checks that the age is at least 18 and the city is 'Sandnes'.

Altering an Existing Table to Add a CHECK Constraint

SQL Command:

ALTER TABLE Persons

ADD CHECK (Age >= 18);

This command adds a check constraint to ensure the `Age` column only allows values 18 or older.

Naming a CHECK Constraint on an Existing Table

ALTER TABLE Persons

ADD CONSTRAINT CHK\_PersonAge CHECK (Age >= 18 AND City = 'Sandnes');

This command adds a named check constraint `CHK\_PersonAge` to the `Persons` table.

Dropping a CHECK Constraint

SQL Command:

ALTER TABLE Persons

DROP CHECK CHK\_PersonAge;

This command drops the `CHK\_PersonAge` constraint from the `Persons` table.

Default Constraint in MySQL

Concept:

- The DEFAULT constraint is used to set a default value for a column.

- The default value will be used if no other value is specified when inserting a new record.

Example:

Table: Persons

| ID | LastName | FirstName | Age | City |

|----|----------|-----------|-----|----------|

| 1 | Smith | John | 30 | Sandnes |

| 2 | Johnson | Jane | 25 | Oslo |

| 3 | Williams | Emily | 22 | Bergen |

Creating a Default Constraint on Table Creation

CREATE TABLE Persons (

ID int NOT NULL,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

City varchar(255) DEFAULT 'Sandnes'

);

This sets the default value for the `City` column as 'Sandnes'.

Altering an Existing Table to Add a Default Constraint

SQL Command:

ALTER TABLE Persons

ALTER City SET DEFAULT 'Sandnes';

This command sets the default value for the `City` column to 'Sandnes'.

Dropping a Default Constraint

SQL Command:

ALTER TABLE Persons

ALTER City DROP DEFAULT;

This command removes the default constraint from the `City` column.

Creating and Managing Indexes in MySQL

Concept:

- Indexes are used to retrieve data more quickly.

- They are created on columns that will be frequently searched against.

- MySQL supports both regular indexes and unique indexes, which prevent duplicate values.

Example:

Table: Persons

| ID | LastName | FirstName | Age |

|----|----------|-----------|-----|

| 1 | Smith | John | 30 |

| 2 | Johnson | Jane | 25 |

| 3 | Williams | Emily | 22 |

Creating an Index

CREATE INDEX idx\_lastname

ON Persons (LastName);

This creates an index on the `LastName` column, which can speed up searches based on last names.

**Creating a Unique Index**

CREATE UNIQUE INDEX idx\_fullname

ON Persons (LastName, FirstName);

This creates a unique index on the combination of `LastName` and `FirstName`, ensuring that no two persons have the same full name.

**Dropping an Index**

SQL Command:

ALTER TABLE Persons

DROP INDEX idx\_lastname;

This command drops the `idx\_lastname` index from the `Persons` table.

**AUTO\_INCREMENT in MySQL**

Concept:

- The `AUTO\_INCREMENT` keyword automatically generates a unique value for a column.

- Typically used for primary key columns to ensure each record has a unique identifier.

Example:

Table: Persons

| PersonID | LastName | FirstName | Age |

|----------|----------|-----------|-----|

| 1 | Smith | John | 30 |

| 2 | Johnson | Jane | 25 |

| 3 | Williams | Emily | 22 |

Creating an AUTO\_INCREMENT Column

CREATE TABLE Persons (

PersonID int NOT NULL AUTO\_INCREMENT,

LastName varchar(255) NOT NULL,

FirstName varchar(255),

Age int,

PRIMARY KEY (PersonID)

);

This SQL command creates an `AUTO\_INCREMENT` primary key column `PersonID`.

Setting the Start Value for AUTO\_INCREMENT

ALTER TABLE Persons AUTO\_INCREMENT = 100;

This command sets the starting value for the `PersonID` column to 100.

**Working with Dates in MySQL**

Concept:

- MySQL provides several data types for handling date and time values. The most commonly used types are:

- `DATE`: Stores dates in the format `YYYY-MM-DD`.

- `DATETIME`: Stores date and time in the format `YYYY-MM-DD HH:MM:SS`.

- `TIMESTAMP`: Stores date and time in the same format as `DATETIME`, but with a range of `1970-01-01 00:00:01` UTC to `2038-01-19 03:14:07` UTC.

- `TIME`: Stores time in the format `HH:MM:SS`.

- `YEAR`: Stores a year in 2-digit or 4-digit format.

Creating a Table with Date and Time Columns

Table: Orders

| OrderID | ProductName | OrderDate | DeliveryTime |

|---------|-------------|---------------------|--------------|

| 1 | Geitost | 2024-09-25 | 13:45:30 |

| 2 | Camembert | 2024-09-26 | 09:30:00 |

| 3 | Mascarpone | 2024-09-27 | 17:15:45 |

CREATE TABLE Orders (

OrderID int NOT NULL AUTO\_INCREMENT,

ProductName varchar(255) NOT NULL,

OrderDate DATE,

DeliveryTime TIME,

PRIMARY KEY (OrderID)

);

Date Functions in MySQL

MySQL offers several functions to manipulate and query date and time data. Here are some of the most common ones:

1. `CURDATE()`

Returns the current date.

SELECT CURDATE();

-- Output: 2024-09-25

2. `CURTIME()`

Returns the current time.

SELECT CURTIME();

-- Output: 13:45:30

3. `NOW()`

Returns the current date and time.

SELECT NOW();

-- Output: 2024-09-25 13:45:30

4. `DATE()`

Extracts the date part from a `DATETIME` or `TIMESTAMP` value.

SELECT DATE('2024-09-25 13:45:30');

-- Output: 2024-09-25

5. `DATE\_FORMAT()`

Formats a date value according to the specified format.

SELECT DATE\_FORMAT(OrderDate, '%W, %M %d, %Y') AS formatted\_date

FROM Orders;

-- Output:

-- 'Wednesday, September 25, 2024'

-- 'Thursday, September 26, 2024'

6. `DATEDIFF()`

Returns the number of days between two date values.

SELECT DATEDIFF('2024-09-30', '2024-09-25') AS days\_difference;

-- Output: 5

7. `DATE\_ADD()` and `DATE\_SUB()`

Adds or subtracts a specified time interval to/from a date.

SELECT DATE\_ADD('2024-09-25', INTERVAL 7 DAY) AS next\_week;

-- Output: 2024-10-02

SELECT DATE\_SUB('2024-09-25', INTERVAL 7 DAY) AS last\_week;

-- Output: 2024-09-18

8. `DAY()`, `MONTH()`, `YEAR()`

Extracts the day, month, or year from a date.

SELECT DAY('2024-09-25') AS day;

-- Output: 25

SELECT MONTH('2024-09-25') AS month;

-- Output: 9

SELECT YEAR('2024-09-25') AS year;

-- Output: 2024

#### 9. `DAYNAME()` and `MONTHNAME()`

Returns the name of the day or month for a given date.

SELECT DAYNAME('2024-09-25') AS day\_name;

-- Output: 'Wednesday'

SELECT MONTHNAME('2024-09-25') AS month\_name;

-- Output: 'September'

10. `STR\_TO\_DATE()`

Converts a string into a date using the specified format.

SELECT STR\_TO\_DATE('25,09,2024', '%d,%m,%Y') AS formatted\_date;

-- Output: 2024-09-25

Using Dates in Queries

1. Selecting Records with a Specific Date:

SELECT \* FROM Orders

WHERE OrderDate = '2024-09-25';

This query selects all orders placed on September 25, 2024.

2. Selecting Records with Date Ranges:

SELECT \* FROM Orders

WHERE OrderDate BETWEEN '2024-09-25' AND '2024-09-27';

This query selects all orders placed between September 25 and September 27, 2024.

3. Selecting Records by Month or Year:

SELECT \* FROM Orders

WHERE YEAR(OrderDate) = 2024;

This query selects all orders placed in the year 2024.

Date Arithmetic

MySQL allows arithmetic operations on date and time values using `INTERVAL`.

1. Adding Days to a Date:

SELECT OrderDate, DATE\_ADD(OrderDate, INTERVAL 10 DAY) AS DeliveryDate

FROM Orders;

This query calculates the delivery date by adding 10 days to the `OrderDate`.

2. Subtracting Months from a Date:

SELECT OrderDate, DATE\_SUB(OrderDate, INTERVAL 2 MONTH) AS OrderPlacementDate

FROM Orders;

This query finds the order placement date by subtracting 2 months from the `OrderDate`.

Date Comparison

MySQL provides operators to compare date values directly.

1. Finding Orders Placed Before a Certain Date:

SELECT \* FROM Orders

WHERE OrderDate < '2024-09-25';

This query selects all orders placed before September 25, 2024.

2. Finding Orders Placed After a Certain Date:

SELECT \* FROM Orders

WHERE OrderDate > '2024-09-25';

This query selects all orders placed after September 25, 2024.

Formatting Dates for Display

Using `DATE\_FORMAT()`, dates can be formatted in various ways for readability.

SELECT OrderDate,

DATE\_FORMAT(OrderDate, '%W, %M %d, %Y') AS formatted\_date

FROM Orders;

- `%W`: Full weekday name (e.g., 'Wednesday')

- `%M`: Full month name (e.g., 'September')

- `%d`: Day of the month, two digits (e.g., '25')

- `%Y`: Four-digit year (e.g., '2024')

Output:

| OrderDate | formatted\_date |

|------------|------------------------------|

| 2024-09-25 | Wednesday, September 25, 2024|

| 2024-09-26 | Thursday, September 26, 2024 |

| 2024-09-27 | Friday, September 27, 2024 |

Storing Only the Date Part of a Date-Time Value

Sometimes you may want to store only the date part of a `DATETIME` or `TIMESTAMP` value.

SELECT DATE(NOW()) AS today\_date;

This query extracts only the date part of the current date-time value.This section now includes detailed explanations, examples, and practical use cases for working with dates in MySQL.