**Intro my sql**

**Overview categories**

**Ddl ,Dml,Dql**

**Datatypes-after dummy database**

**Operators**

**Clauses**

**Single row and multi row functions**

**Constraints**

**Joins**

**Indexes**

**Tcl**

**Dcl**

**Stored procedures**

**Views**

**Date methods**

**Keys**

**Normalization -6**

**Triggers**

**Window functions**

**Oncascade ondelete**

**If else**

**If null**

**Database**

It is a **collection of structured or unstructured data** in a format that can be easily accessed digitally and controlled by a dbms

### 

### 

### 

### 

### 

### 

### **DBMS (Database Management System)**

A DBMS is a software used to **create, manage, and operate** databases by providing tools for data storage, retrieval, and security.

**Types of databases**

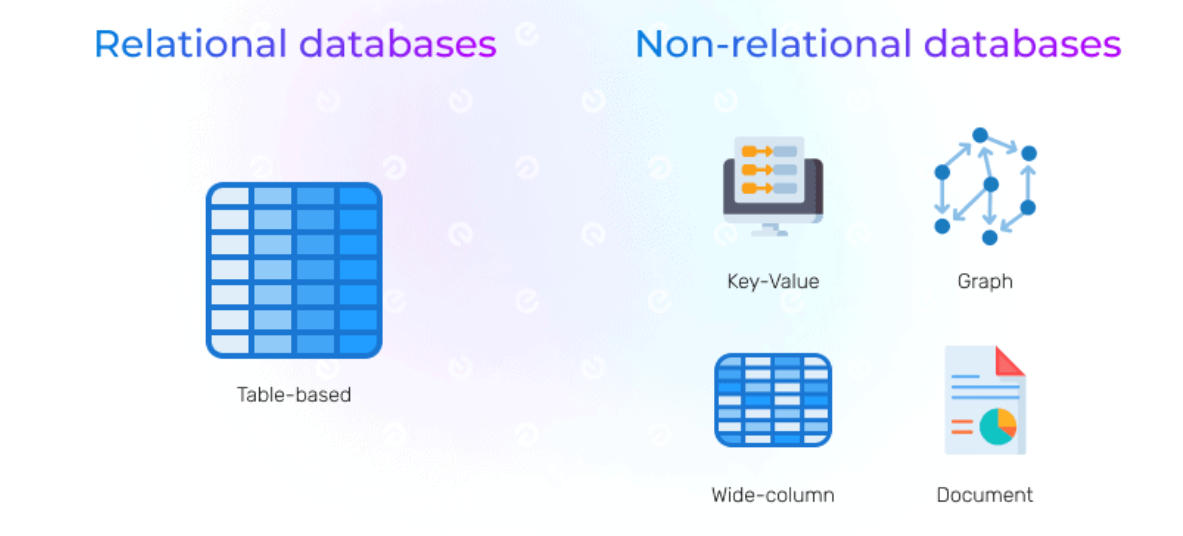
1) **Relational databases**:- it is a database where the data is stored in tables in a structured format

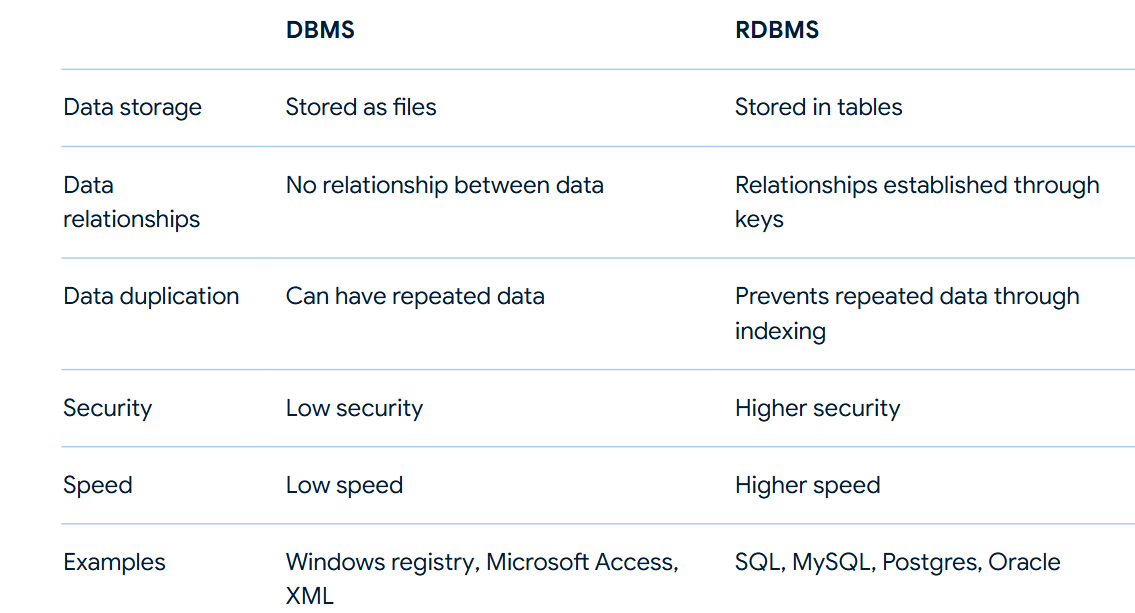
Ex:- mysql , postgresql, oracle database,amazon rds

2) **Non-relational databases**:- it is a database where the data is stored in unstructured way

such as documents, key-value pairs, graphs, or wide-column stores.

Ex:- mongodb, no sql, neo4



****

**Types of dbms**

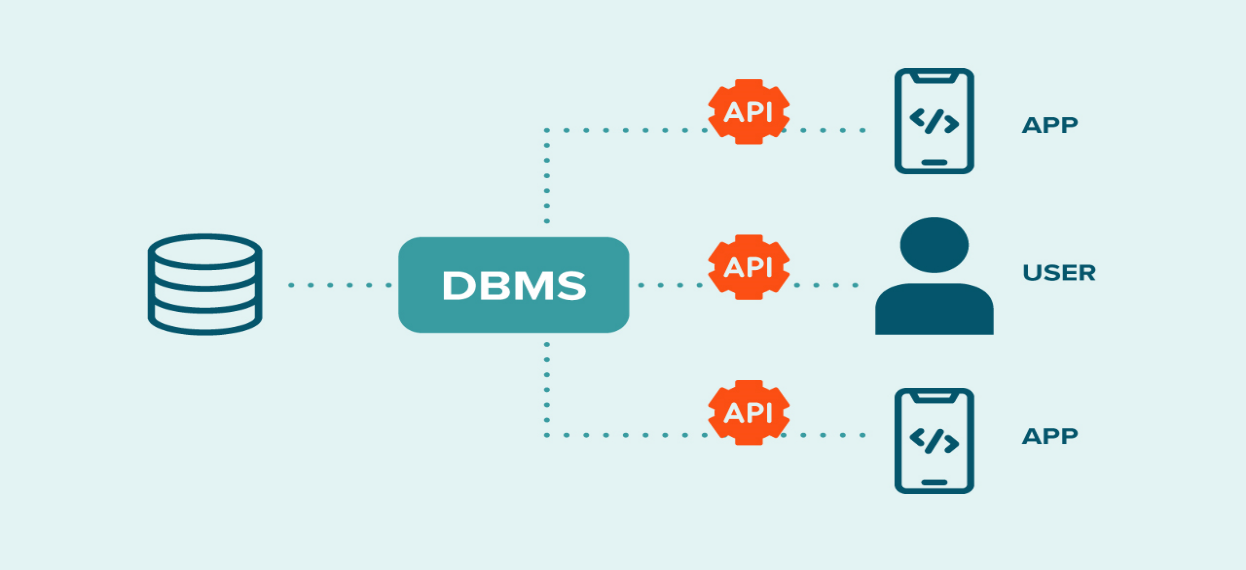
Rdbms

Ordbms

Fms- file management system

Hms- heirerchial management system

Nms-network management system

DBMS Tutorial – Learn Database Management System - GeeksforGeeksDBMS Tutorial – Learn Database Management System - GeeksforGeeks

**Relational Database**: A collection of related tables.

**Table**: collection of rows and columns (like an Excel sheet).

**Column (Field)**: collection of field names

A single attribute of the table.

* Example: id, name, branch, age

**Row (Record)**:collection of field values

One complete set of related data.

* Example (student): 1, "Amit", "CS", 20

**Features of rdbms**

Easy to access and manipulate data

Less redundancy – duplication of data

Guarantee data quality

Supports data sharing , transaction and security

**Rdbms softwares**

| **RDBMS** | **Provider** |
| --- | --- |
| **MySQL** | Oracle |
| **Oracle Database** | Oracle |
| **PostgreSQL** | PostgreSQL Global Development Group |
| **SQL Server** | Microsoft |
| **DB2** | IBM |
| **Amazon RDS** | AWS (cloud-based managed service) |

**Database can be deployed in two way**

On premises—deployed in the server which is managed by client

On cloud--- deployed in the server where data is managed by cloud service providers

**Sql (structured query language)**

Sql is statically typed programming language used to interact with relational database

Used to query and retrieve data from the database

=> it is a language used to communicate with the mysql.

=> user communicates with mysql by sending commands called queries.

=> a query is a command/request/question submitted to mysql

to perform some operation over db.

=> sql commands are categorized into 5 sub languages.

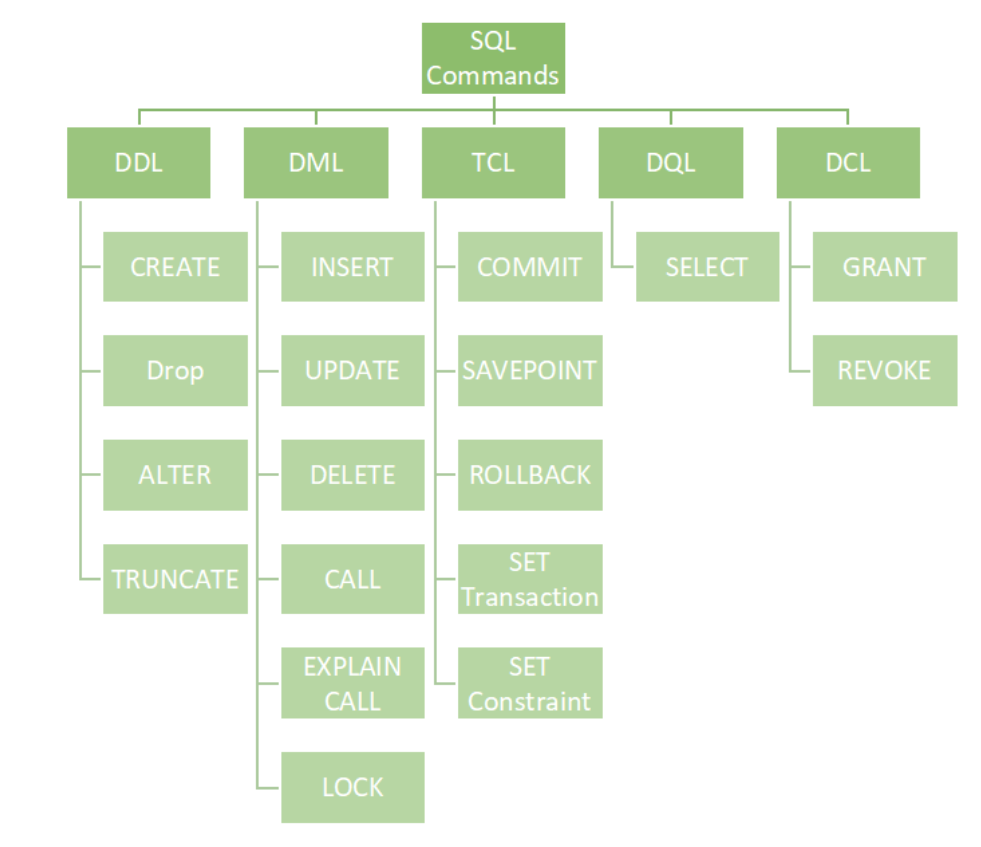
1 DDL (Data Definition Lang)

2 DML (Data Manipulation Lang)

3 DQL (Data Query Lang)

4 DCL (Data Control Lang)

5 TCL (Transaction Control Lang)

****

**Mysql;**

it is a opensource rdbms(relational database management system) based on structured query language used to handle wide range of database applications

* **SQL**: A standard language for interacting with databases, used to perform tasks such as querying data, updating records, and managing database structures.

**Installing MySQL on Windows**

1. **Download MySQL Installer**:
   * Go to the [MySQL Community Downloads](https://dev.mysql.com/downloads/installer/) page.
   * Choose the MySQL Installer for Windows.
2. **Run the Installer**:
   * Double-click the downloaded **.msi** file to start the installation process.
3. **Choose Setup Type**:
   * Choose the setup type that suits your needs: Developer Default, Server only, Client only, Full, or Custom. For most users, ‘Developer Default’ is recommended.
4. **Check Requirements**:
   * The installer will check for any missing requirements. Install any required software as prompted.
5. **Installation**:
   * Click ‘Execute’ to download and install the necessary MySQL components.
6. **Configuration**:
   * After installation, the MySQL installer will guide you through configuration steps:
     + **High Availability**: Select ‘Standalone MySQL Server / Classic MySQL Replication.’
     + **Type and Networking**: Choose ‘Development Computer’ or the appropriate setting for your setup. The default port is 3306.
     + **Authentication**: Set up the root password and create additional user accounts if needed.
     + **Windows Service**: Configure MySQL to run as a Windows service if desired.
7. **Complete the Installation**:
   * Click ‘Execute’ to apply the configuration settings.
   * Click ‘Finish’ once the configuration is complete.
8. **Verify Installation**:
   * Open the MySQL Command Line Client from the Start menu and log in with the root password to verify the installation.

**Connecting to MySQL Shell**

Before you can create a database, you need to connect to the MySQL server. This is typically done using the MySQL command-line client or a graphical tool like MySQL Workbench.

**MySQL Shell Commands**

* \sql → Run SQL queries
* \**connect root@localhost:3306** → Connect to MySQL
* \! cls; → Clear the terminal screen

**Viewing Databases**

To see a list of all databases on the MySQL server, use the SHOW DATABASES command.

**SHOW DATABASES;**

**Creating a Database**

Once connected to the MySQL server, you can create a database using the CREATE DATABASE statement.

Syntax

**CREATE DATABASE database\_name;**

**Selecting a Database**

After creating a database, you need to select it to perform operations such as creating tables, inserting data, and running queries.

**USE database\_name;**

**Checking the Current Database**

To check which database you are currently using, you can run:

**SELECT DATABASE();**

**Dropping a Database**

To delete a database, use the DROP DATABASE statement. Be cautious, as this action is irreversible and will permanently remove the database and all its data.

**DROP DATABASE database\_name;**

**Overview categories**

**SQL Categories and Commands**

SQL (Structured Query Language) is divided into several categories based on the type of operations they perform. Below are the major categories along with their respective commands and descriptions.

**1. Data Definition Language (DDL)**

DDL commands are used to define and manage database structures.

* **CREATE**: Used for creating tables, databases, indexes, and users.
* **ALTER**: Modifies table structures.
* **DROP**: Deletes tables, columns, databases, indexes, and users.
* **TRUNCATE**: Removes all data from a table.
* **RENAME**: Used to rename tables and columns.

**Examples:**

**CREATE TABLE employees (**

**id INT PRIMARY KEY,**

**name VARCHAR(50),**

**salary DECIMAL(10,2)**

**);**

ALTER TABLE employees ADD COLUMN department VARCHAR(50);

DROP TABLE employees;

TRUNCATE TABLE employees;

RENAME TABLE employees TO staff;

**2. Data Manipulation Language (DML)**

DML commands are used to manipulate data within database objects.

* **INSERT**: Adds new data into a table (single row and multiple rows).
* **UPDATE**: Modifies existing records using the SET clause.
* **DELETE**: Removes specific data from a table (faster than TRUNCATE).

**Examples:**

INSERT INTO employees (id, name, salary) VALUES (1, 'John Doe', 50000);

UPDATE employees SET salary = 55000 WHERE id = 1;

DELETE FROM employees WHERE id = 1;

**3. Data Control Language (DCL)**

DCL commands are used to control access to data in the database.

* **GRANT**: Gives permissions to users.
* **REVOKE**: Removes permissions from users.

**Examples:**

GRANT SELECT, INSERT ON employees TO user1;

REVOKE INSERT ON employees FROM user1;

**4. Transaction Control Language (TCL)**

TCL commands manage database transactions.

* **COMMIT**: Saves changes permanently.
* **ROLLBACK**: Undoes changes.
* **SAVEPOINT**: Creates a checkpoint within a transaction to which changes can be rolled back.

**Examples:**

START TRANSACTION;

INSERT INTO employees (id, name, salary) VALUES (2, 'Jane Doe', 60000);

SAVEPOINT sp1;

UPDATE employees SET salary = 65000 WHERE id = 2;

ROLLBACK TO sp1; -- Undo update but keep insert

COMMIT;

**Auto-commit Mode:**

* **AUTOCOMMIT = ON**: Each statement is committed automatically.
* **AUTOCOMMIT = OFF**: Transactions must be manually committed.

**5. Data Query Language (DQL)**

DQL is used for retrieving data from the database.

* **SELECT**: Fetches data from one or more tables.

**Example:**

SELECT \* FROM employees;

SELECT name, salary FROM employees WHERE salary > 50000;

COPY THE DUMMY DATABASE DATA

DROP database if exists practice;

create database practice;

use practice;

CREATE TABLE dept (

deptno DECIMAL(2,0) PRIMARY KEY,

dname VARCHAR(14) DEFAULT NULL,

loc VARCHAR(13) DEFAULT NULL

);

CREATE TABLE emp (

empno DECIMAL(4,0) NOT NULL,

ename VARCHAR(10) DEFAULT NULL,

job VARCHAR(9) DEFAULT NULL,

mgr DECIMAL(4,0) DEFAULT NULL,

hiredate DATE DEFAULT NULL,

sal DECIMAL(7,2) DEFAULT NULL,

comm DECIMAL(7,2) DEFAULT NULL,

deptno DECIMAL(2,0),

FOREIGN KEY (deptno) REFERENCES dept(deptno)

);

-- Insert data into the dept table first

INSERT INTO dept VALUES ('10', 'ACCOUNTING', 'NEW YORK');

INSERT INTO dept VALUES ('20', 'RESEARCH', 'DALLAS');

INSERT INTO dept VALUES ('30', 'SALES', 'CHICAGO');

INSERT INTO dept VALUES ('40', 'OPERATIONS', 'BOSTON');

-- Now insert data into the emp table

INSERT INTO emp VALUES ('7369', 'SMITH', 'CLERK', '7902', '1980-12-17', '800.50', NULL, '20');

INSERT INTO emp VALUES ('7499', 'ALLEN', 'SALESMAN', '7698', '1981-02-20', '1600.00', '300.00', '30');

INSERT INTO emp VALUES ('7521', 'WARD', 'SALESMAN', '7698', '1981-02-22', '1250.23', '500.00', '30');

INSERT INTO emp VALUES ('7566', 'JONES', 'MANAGER', '7839', '1981-04-02', '2975.00', NULL, '20');

INSERT INTO emp VALUES ('7654', 'MARTIN', 'SALESMAN', '7698', '1981-09-28', '1250.81', '1400.00', '30');

INSERT INTO emp VALUES ('7698', 'BLAKE', 'MANAGER', '7839', '1981-05-01', '2850.56', NULL, '30');

INSERT INTO emp VALUES ('7782', 'CLARK', 'MANAGER', '7839', '1981-06-09', '2450.00', NULL, '10');

INSERT INTO emp VALUES ('7788', 'SCOTT', 'ANALYST', '7566', '1982-12-09', '3000.61', NULL, '20');

INSERT INTO emp VALUES ('7839', 'KING', 'PRESIDENT', NULL, '1981-11-17', '5000.00', NULL, '10');

INSERT INTO emp VALUES ('7844', 'TURNER', 'SALESMAN', '7698', '1981-09-08', '1500.40', '0.00', '30');

INSERT INTO emp VALUES ('7876', 'ADAMS', 'CLERK', '7788', '1983-01-12', '1100.00', NULL, '20');

INSERT INTO emp VALUES ('7900', 'JAMES', 'CLERK', '7698', '1981-12-03', '950.20', NULL, '30');

INSERT INTO emp VALUES ('7902', 'FORD', 'ANALYST', '7566', '1981-12-03', '3000.67', NULL, '20');

INSERT INTO emp VALUES ('7934', 'MILLER', 'CLERK', '7782', '1982-01-23', '1300.00', NULL, 10);

**Datatypes**

**INT**

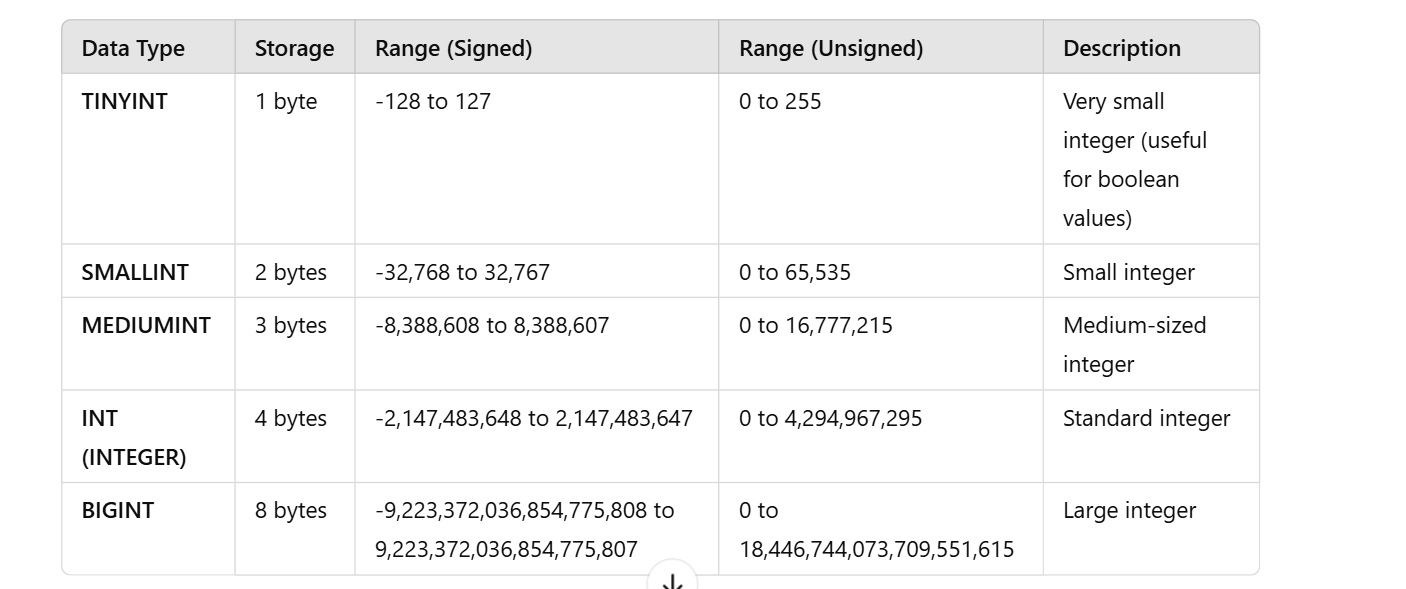
* **Purpose:** Stores whole numbers (both positive and negative) without decimals.
* **Syntax: INT**

**Example:**

**CREATE TABLE employees (**

**id INT NOT NULL**

**);**

****

**VARCHAR**

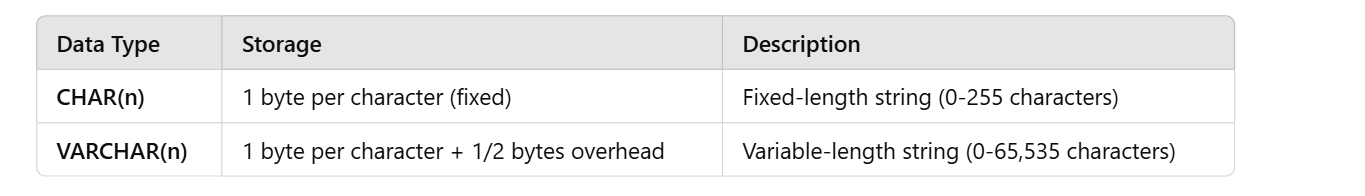
* **Purpose**: Stores variable-length strings**.**
* **Syntax:** VARCHAR(size) where size defines the maximum number of characters.

**Example:**

**CREATE TABLE employees (**

**name VARCHAR(100) NOT NULL**

**);**

****

📌 Example:char(10)  
 Imagine you have a 10-slot box for words.

* If you store "YES", it still occupies all 10 slots (extra space wasted).
* If you store "HELLOWORLD", it fits exactly

📌 **Example:**

Imagine a **flexible box** that expands based on word length.

* If you store "YES", it **uses only 3 slots** instead of 10.
* If you store "HELLOWORLD", it **uses 10 slots**.

**DECIMAL**

* **Purpose**: Stores numbers with fixed precision and scale (e.g., monetary values).
* **Syntax**: DECIMAL(precision, scale) where precision is the total number of digits and scale is the number of digits after the decimal point.

**Example:**

**CREATE TABLE employees (**

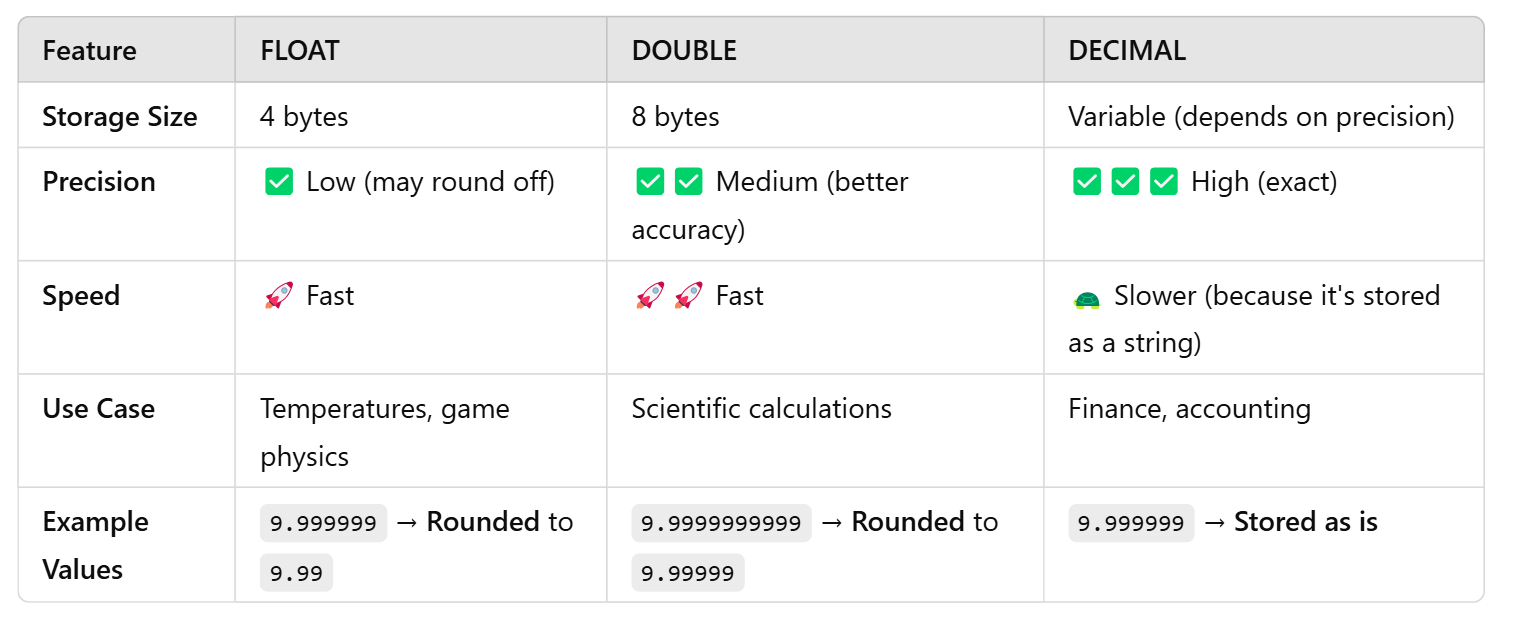
**salary DECIMAL(10, 2) CHECK (salary >= 0)**

**);**

****

🔹 Explanation:

* FLOAT(5,2) → Maximum 5 digits total, 2 digits after decimal (e.g., 999.99).
* DOUBLE(8,3) → Maximum 8 digits, 3 decimal places (e.g., 12345.678).
* DECIMAL(10,2) → **10 total digits**, **2 decimal places**.



**DATE**

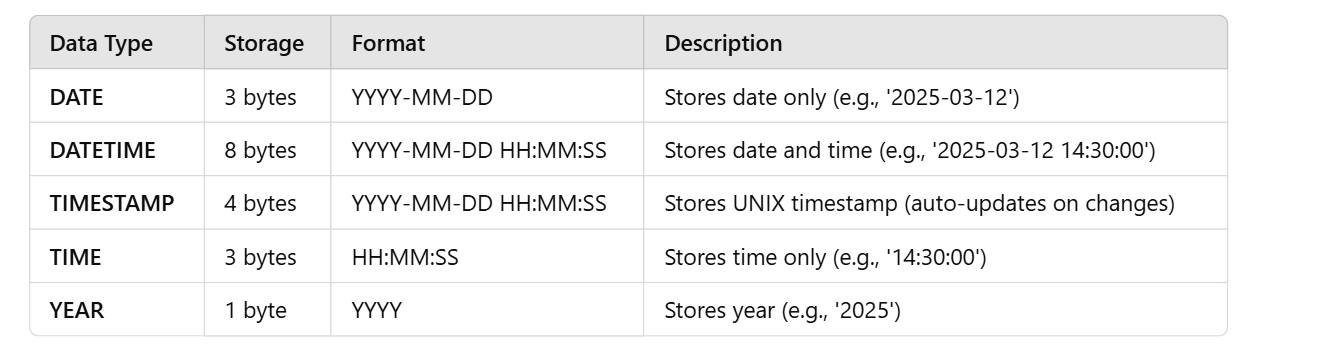
* **Purpose: Stores dates in the format YYYY-MM-DD.**
* **Syntax: DATE**

**Example:**

**CREATE TABLE employees (**

**hire\_date DATE**

**);**

****

**TIMESTAMP**

* **Purpose:** Stores both date and time in the format YYYY-MM-DD HH:MM:SS.
* **Syntax:** TIMESTAMP

**Example:**

**CREATE TABLE employees (**

**created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP**

**);**

**BOOLEAN**

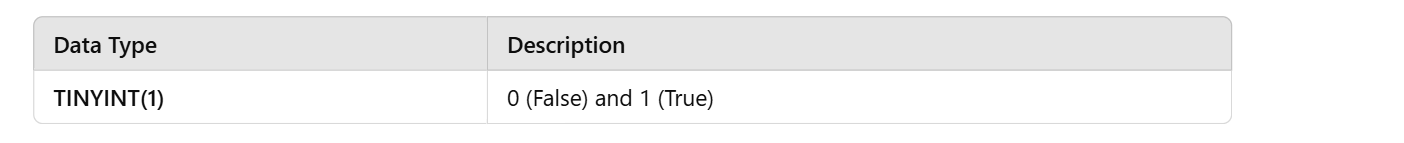
* **Purpose:** Stores a value of TRUE or FALSE.
* **Syntax**: BOOLEAN (often stored as TINYINT(1) in some databases).

**Example:**

**CREATE TABLE employees (**

**is\_active BOOLEAN DEFAULT TRUE**

**);**

****

**TEXT**

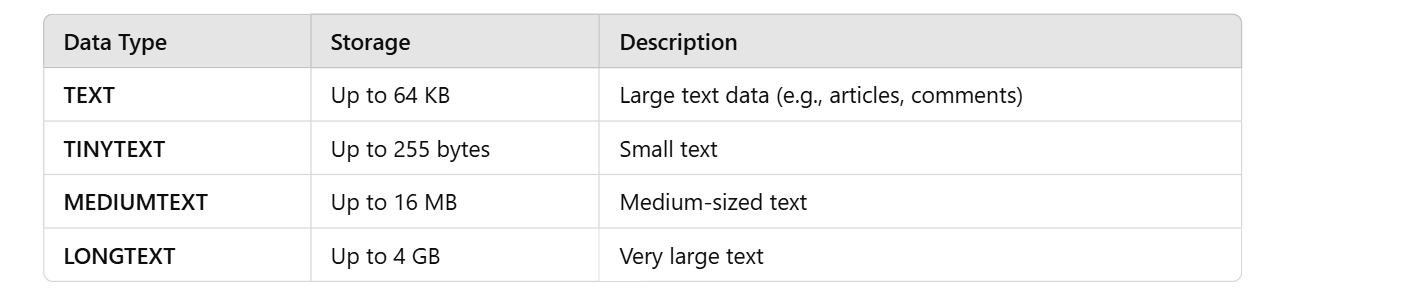
* **Purpose:** Stores large amounts of text data.
* **Syntax:** TEXT

**Example:**

**CREATE TABLE employees (**

**description TEXT**

**);**

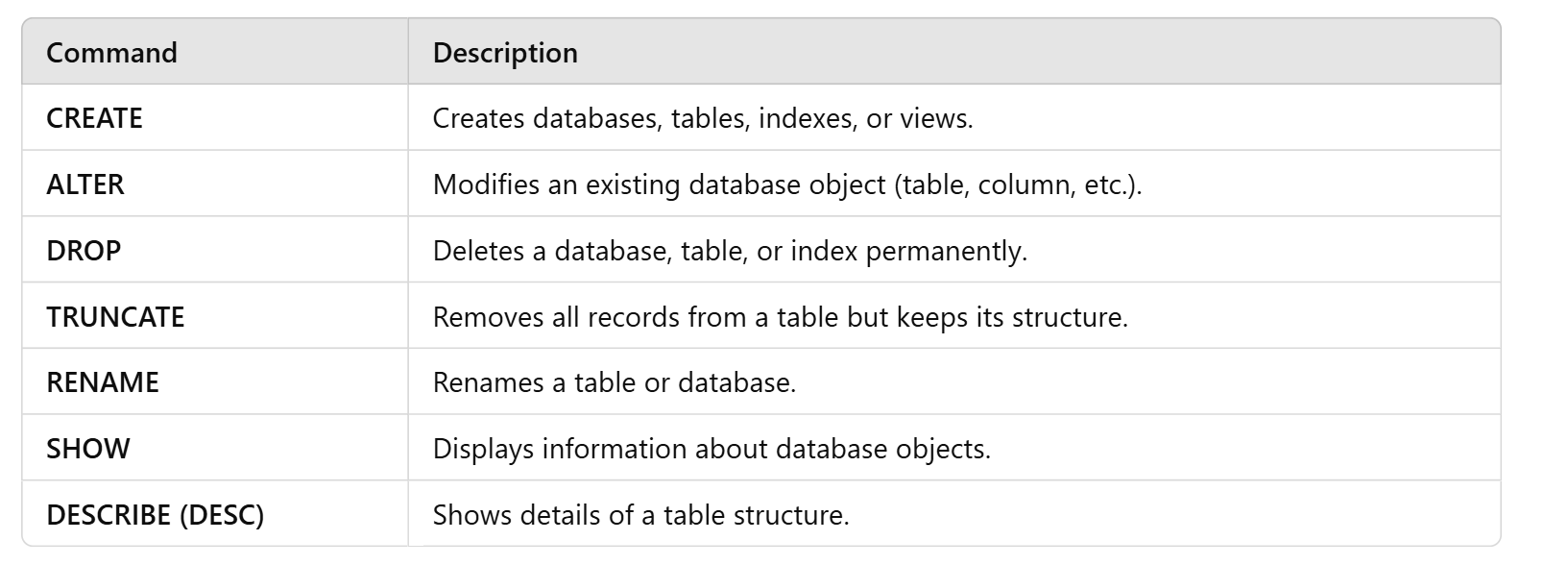
****

**How to handle tables**

**Data definition language Queries**

DDL (Data Definition Language) is used to define and manage the structure of a database. These queries create, modify, and delete databases, tables, and other objects like indexes.

DDL commands do not modify data but change the schema of the database.



**Creating a Table**

Creates a new table with specified columns

Syntax

**CREATE TABLE table\_name ( column1 datatype constraints, column2 datatype constraints);**

CREATE TABLE users (

id INT,

username VARCHAR(50),

password VARCHAR(50),

email VARCHAR(100),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

To show the description of the table

**Desc tablename;**

**Altering a Table**

**1)****Adding a Column**:-Adds a new column to an existing table

**ALTER TABLE table\_name ADD column\_name datatype constraints;**

**2)Modifying a Column**:-Changes the data type and/or constraints of an existing column.

**ALTER TABLE table\_name MODIFY COLUMN column\_name new\_datatype new\_constraints;**

**3)** **Dropping a Column :-**Removes a column from an existing table.

**ALTER TABLE table\_name DROP COLUMN column\_name;**

**4)** **Renaming a Column:-** Renames an existing column

**ALTER TABLE table\_name RENAME COLUMN colname to newcolname;**

**5) Add a constraint**

**ALTER TABLE table\_name ADD CONSTRAINT colname CHECK (salary >= 3000);**

**Renaming a Table**

Changes the name of an existing table.

**ALTER TABLE old\_table\_name RENAME TO new\_table\_name;**

**Rename oldtable to newtable;**

**Dropping a Table**

Deletes the table and all of its data permanently. Use with caution.

**DROP TABLE table\_name;**

**Truncate:**

It removes the data in the table

**Truncate table tname;**

**Viewing Tables**

To see a list of all tables on the MySQL server, use the SHOW tables command.

**SHOW TABLES;**

**Data manipulation language Queries**

DML commands are used to manipulate data within database objects.

* **INSERT**: Adds new data into a table (single row and multiple rows).
* **UPDATE**: Modifies existing records using the SET clause.
* **DELETE**: Removes specific data from a table (faster than TRUNCATE).

**INSERT**

The **INSERT** statement is used to add new rows (single row and multiple rows) to a table.

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

Multiple Rows:

**INSERT INTO** employees (first\_name, last\_name, department)

**VALUES** ('Jane', 'Doe', 'HR'), ('Mark', 'Smith', 'IT');

**UPDATE**

The **UPDATE** statement is used to modify existing records in a table.

**UPDATE** table\_name **SET** column1 = ‘value1’, column2 = ‘value2’, ...

**WHERE** condition=’value’;

**DELETE**

The **DELETE** statement is used to remove existing records from a table.

DELETE FROM table\_name

WHERE condition=’value’;

**Caution:** Omitting the **WHERE** clause will delete all rows in the table.

**Data Query Language (DQL)**

DQL is used for querying and fetching data from a database.

**SELECT Statement**

The **SELECT** statement is used to retrieve data from one or more tables.

**SELECT column1, column2 FROM table\_name;**

**Selecting All Columns**

To select all columns from a table, use the asterisk (**\***).

**SELECT \* FROM employees;**

**Using Aliases**

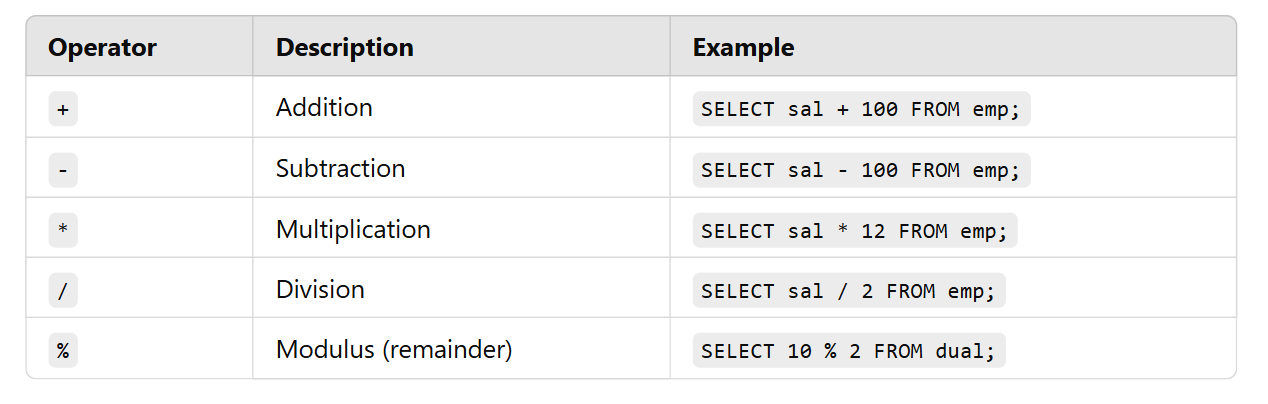
**Aliases can be used to give a table or a column a temporary name.**

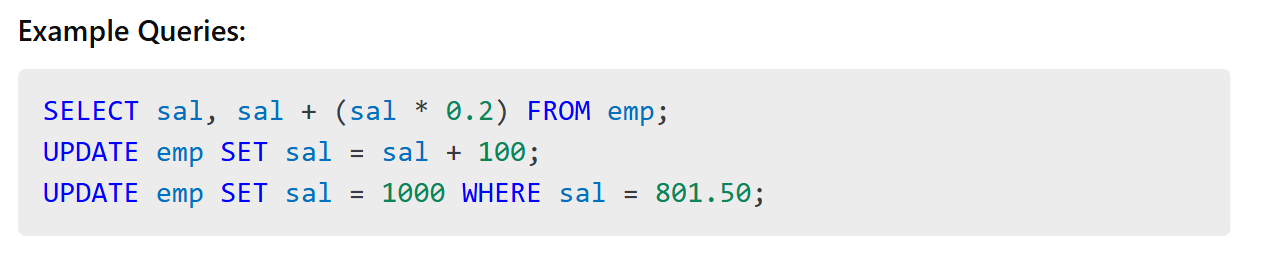
**SELECT first\_name AS fnam FROM employees;**

**MySQL Operators**

**Arithmetic Operators**

* **Addition** (+): Adds two values.
* **Subtraction** (-): Subtracts one value from another.
* **Multiplication** (\*): Multiplies two values.
* **Division** (/): Divides one value by another.
* **Modulus** (%): Returns the remainder of a division.





**Comparison Operators**

* **Equal to (=):** Checks if two values are equal.
* **Not equal to (!= or <>)**: Checks if two values are not equal.
* **Greater than (>):** Checks if a value is greater than another.
* **Less than (<):** Checks if a value is less than another.
* **Greater than or equal to (>=):** Checks if a value is greater than or equal to another.
* **Less than or equal to (<=)**: Checks if a value is less than or equal to another.

****

**Logical Operators**

* **AND (AND): Returns true if both conditions are true.**

And - The AND operator is used to filter records based on more than one condition for different fieldnames

WHERE Country = 'Germany' AND City = ‘newyork’;

* **OR (OR): Returns true if at least one condition is true.**

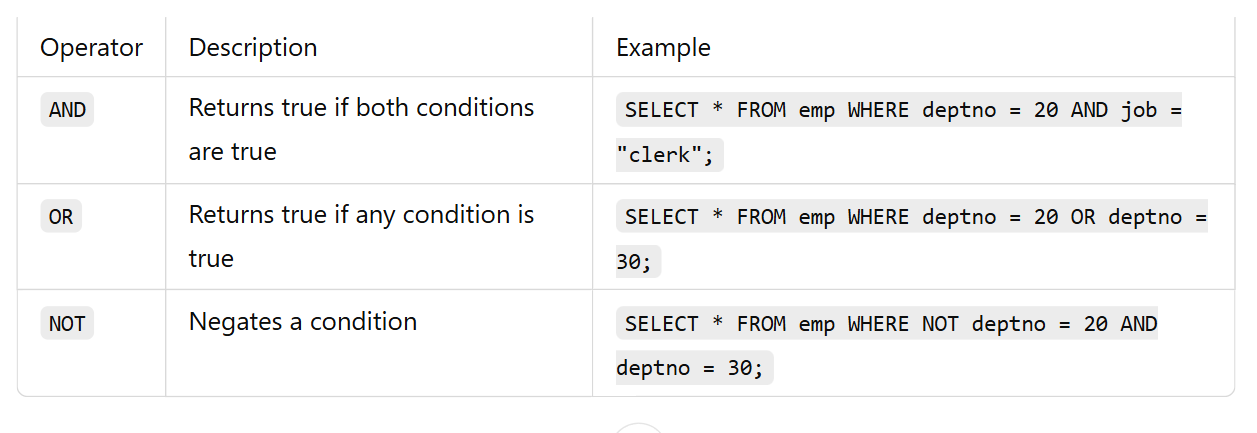
OR -The OR operator is used to filter records based on more than one condition

WHERE Country = 'Germany' OR Country = 'Spain';

* **NOT (NOT): Inverts the truth value.**

NOT - The NOT operator is used in combination with other operators to give the opposite result

WHERE NOT Country = 'Spain';



**Bitwise Operators**

* **AND (&): Bitwise AND.**
  + **Example: SELECT 5 & 3; → 1**
* **OR (|): Bitwise OR.**
  + **Example: SELECT 5 | 3; → 7**
* **XOR (^): Bitwise XOR.**
  + **Example: SELECT 5 ^ 3; → 6**
* **NOT (~): Bitwise NOT.**
  + **Example: SELECT ~5; → -6**
* **Shift left (<<): Shifts bits to the left.**
  + **Example: SELECT 1 << 2; → 4**
* **Shift right (>>): Shifts bits to the right.**
  + **Example: SELECT 4 >> 1; → 2**

**special Operators**

* **IS NULL: Checks if a value is NULL.**

IS NULL The IS NULL operator is used to test for empty values (NULL values).

**Ex:-**

WHERE Address IS NULL;

* **IS NOT NULL: Checks if a value is not NULL.**

IS NOT NULL-The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

**Ex:-**

WHERE Address IS NOT NULL;

* **IN: Checks if a value is within a set of values.**

in() in operators is used

**Ex-**

Where year in (“mad”,2019,”dad”)

* **NOT IN: The NOT IN operator checks if a value does not exist within a set.**

SELECT \* FROM emp WHERE deptno NOT IN (10, 20, 30);

* **BETWEEN: Checks if a value is within a range.**

Between – print the values between the range

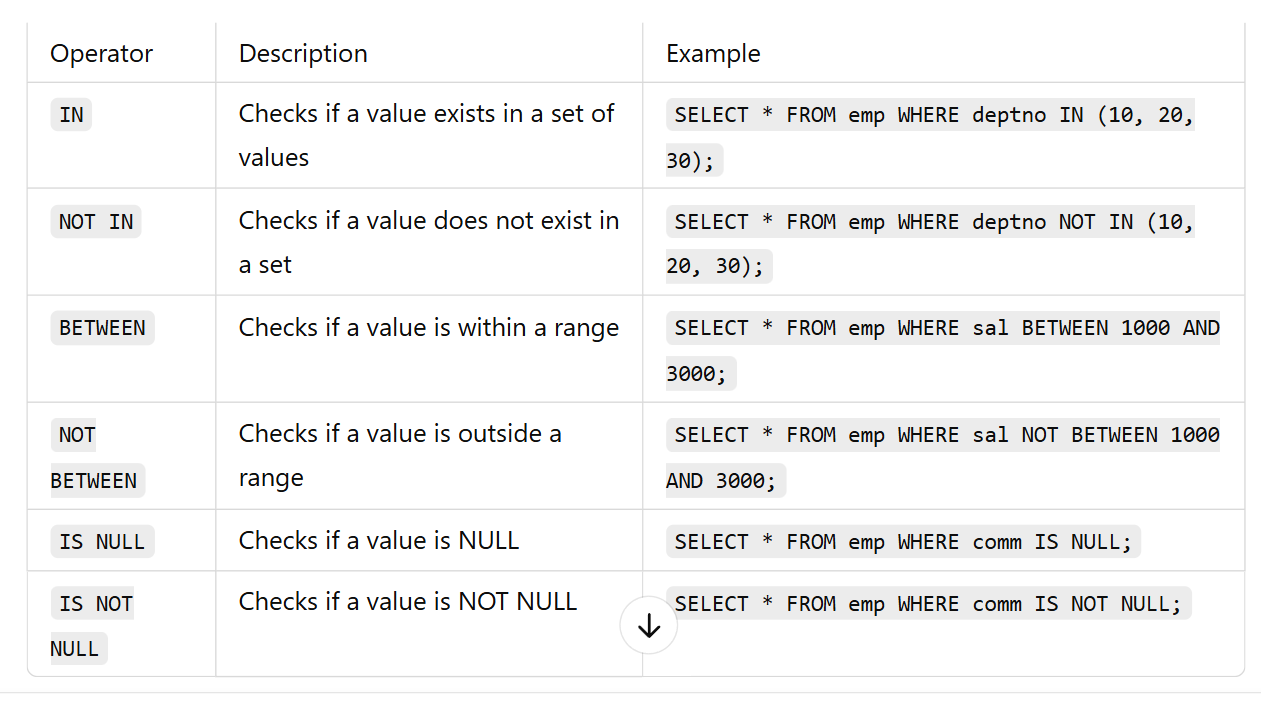
**Ex:-**

Where years between 2019 and 2024

* **NOT BETWEEN: The NOT BETWEEN operator selects values outside a given range.**

**Ex:-**

SELECT \* FROM emp WHERE sal NOT BETWEEN 1000 AND 3000;



**Wildcard Operators (Pattern Matching)**

Wildcards are used with the LIKE operator to match string patterns.

* **LIKE: Checks if a value matches a pattern.**

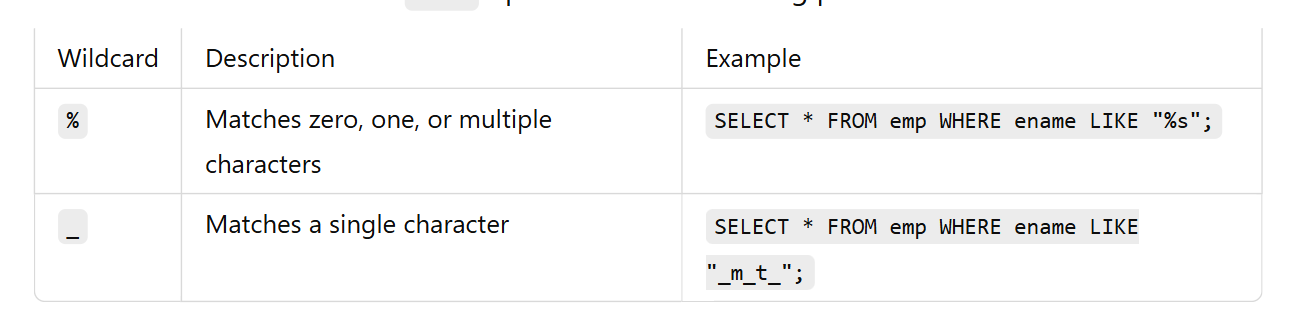
LIKE is used to target pattern

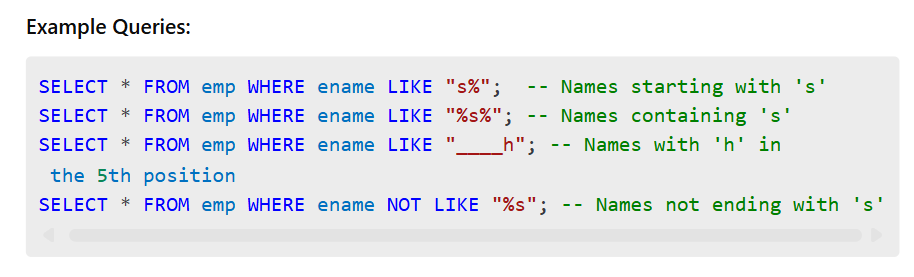
**Ex:-**

SELECT \* FROM Customers

WHERE Country = 'Spain' AND CustomerName like 'g%';

**Wild characters**- used only for strings



****

**MySQL Clauses**

**What are MySQL Clauses?**

Clauses in MySQL are used to filter, sort, and manipulate the data returned by SQL queries. They help refine results and improve query efficiency.

**Types of MySQL Clauses**

1. WHERE Clause
2. DISTINCT Clause
3. HAVING Clause
4. ORDER BY Clause
5. GROUP BY Clause
6. LIMIT Clause
7. OFFSET Clause

**1. WHERE Clause**

The WHERE clause filters records based on specified conditions. It is used in SELECT, UPDATE, and DELETE statements.

**Syntax**

**SELECT column\_name(s) FROM table\_name WHERE condition;**

**Ex:**

SELECT \* FROM emp WHERE sal > 3000;

Usage in UPDATE:

UPDATE emp SET salary = sal+ 5000 WHERE deptno= 10;

Usage in DELETE:

DELETE FROM emp WHERE sal< 2000;

**2. DISTINCT Clause**

The DISTINCT clause removes duplicate records from the result set.

**Syntax:**

**SELECT DISTINCT column\_name FROM table\_name;**

**Example:**

Find unique job titles from the employees table.

SELECT DISTINCT job FROM emp;

**3. HAVING Clause**

The HAVING clause filters results after the GROUP BY operation. It is used to apply conditions on aggregated values.

Syntax:

**SELECT column\_name, aggregate\_function(column\_name) FROM table\_name**

**GROUP BY column\_name**

**HAVING condition;**

Ex:-

SELECT deptno, AVG(sal) AS avg\_salary

FROM emp

GROUP BY deptno

HAVING AVG(sal) > 50000;

**4. ORDER BY Clause**

The ORDER BY clause sorts the result set in ascending (ASC) or descending (DESC) order.

**Syntax:**

**SELECT column\_name(s) FROM table\_name ORDER BY column\_name [ASC|DESC];**

**Example:** Get all employees sorted by salary in descending order.

SELECT \* FROM emp ORDER BY sal DESC;

**Example:** Get employees sorted by department and then by salary

SELECT \* FROM emp ORDER BY deptno ASC, sal DESC;

**5. GROUP BY Clause**

The GROUP BY clause groups rows that have the same values in specified columns and is commonly used with aggregate functions.

**If you use GROUP BY, every selected column must be either:**

1. In GROUP BY
2. OR inside an aggregate function

**Syntax:**

**SELECT column\_name, aggregate\_function(column\_name)**

**FROM table\_name**

**GROUP BY column\_name;**

**Example: Get the total salary per department.**

SELECT deptno, SUM(sal) AS total\_salary

FROM emp

GROUP BY deptno;

### **Multiple columns in GROUP BY**

SELECT dept, city, COUNT(\*)

FROM employees

GROUP BY dept, city;

* First by dept
* Then inside dept → by city

**6. LIMIT Clause**

The LIMIT clause restricts the number of records returned by a query.

**Syntax:**

**SELECT column\_name(s) FROM table\_name LIMIT number;**

**Example: Get the top 3 highest-paid employees.**

SELECT \* FROM emp ORDER BY sal DESC LIMIT 3;

**7. OFFSET Clause**

The OFFSET clause skips a specified number of rows before starting to return rows. It is often used with LIMIT.

**Syntax:**

**SELECT column\_name(s) FROM table\_name LIMIT number OFFSET offset\_value;**

**Example: Get the second highest-paid employee.**

SELECT \* FROM emp ORDER BY sal DESC LIMIT 1 OFFSET 1;

**Example: Get employees ranked 4th to 6th based on salary.**

SELECT \* FROM emp ORDER BY sal DESC LIMIT 3 OFFSET

**Single row and multi row functions**

MySQL provides functions to perform operations on data. These functions are categorized into:

1. **Single-Row Functions** → Operate on each row and return one result per row.
2. **Multi-Row (Aggregate) Functions** → Operate on multiple rows and return a single result.

**1. Single-Row Functions**

Single-row functions return **one result per row** and can be used in SELECT, WHERE, and ORDER BY clauses.

**1.1 String Functions**

**🔹 UPPER()** – Converts a string to uppercase

Syntax:

**SELECT UPPER(column\_name) FROM table\_name;**

Ex:

SELECT UPPER(ename) FROM emp;

If ename = 'john', output → JOHN

**🔹 LOWER() – Converts a string to lowercase**Syntax:

**SELECT LOWER(column\_name) FROM table\_name;**

Ex:

SELECT LOWER(job) FROM emp;

If job = 'MANAGER', output → manager

**🔹 LENGTH() –** Returns the number of characters in a string

Syntax:

**SELECT LENGTH(column\_name) FROM table\_name;**

Ex:

SELECT ename, LENGTH(ename) FROM emp;

If ename = 'Alice', output → 5

**🔹 CONCAT() – Joins two or more strings**

Syntax:

**SELECT CONCAT(string1, string2, ...) FROM table\_name;**

Ex:

SELECT CONCAT(ename, ' - ', job) FROM emp;

If ename = 'John' and job = 'Clerk', output → John - Clerk

**🔹 SUBSTRING() – Extracts a portion of a string**

Syntax:

**SELECT SUBSTRING(column\_name, start\_position, length) FROM table\_name;**

Ex:

SELECT SUBSTRING(ename, 1, 3) FROM emp;

If ename = 'Michael', output → Mic

**1.2 Numeric Functions**

Numeric functions perform calculations on numeric data.

**🔹 ROUND() – Rounds a number to a given decimal places**

Syntax:

**SELECT ROUND(number, decimal\_places);**

Ex:

SELECT ROUND(sal, 2) FROM emp;

If sal = 1234.5678, output → 1234.57

**🔹 CEIL() – Rounds a number up to the nearest integer**

Syntax:

**SELECT CEIL(number);;**

Ex:

SELECT CEIL(sal) FROM emp;

If sal = 1234.45, output → 1235

**🔹 FLOOR() – Rounds a number down to the nearest integer**

Syntax:

**SELECT FLOOR(number);**

Ex:

SELECT FLOOR(sal) FROM emp;

If sal = 1234.99, output → 1234

**🔹 ABS()** – Returns absolute value of a number

Ex-

SELECT ABS(-100) FROM dual;

**🔹 MOD()** – Returns remainder of division

Ex-

SELECT MOD(10, 3);

**1.3 Date & Time Functions**

Date functions operate on date and time values.

**🔹 CURDATE() – Returns the current date**

Syntax**:**

**SELECT CURDATE();**

**Ex:**

SELECT CURDATE();

Output → 2025-03-22 (Current date)

**🔹 CURTIME() – Returns the current time**

Syntax:

**SELECT CURTIME();**

Example:

SELECT CURTIME();

Output → 14:30:45 (Current time)

**🔹 YEAR(), MONTH(), DAY() – Extracts year, month, or day from a date**

Syntax:

**SELECT YEAR(date\_column), MONTH(date\_column), DAY(date\_column) FROM table\_name;**

Example:

SELECT YEAR(hiredate), MONTH(hiredate), DAY(hiredate) FROM emp;

If hiredate = '2015-06-12', output → 2015 | 6 | 12

**🔹 DATEDIFF() – Returns the difference between two dates**

Syntax:

**SELECT DATEDIFF(date1, date2);**

Example:

SELECT DATEDIFF(CURDATE(), hiredate) FROM emp;

Returns **number of days** between hire date and today.

**🔷 2. Multi-Row (Aggregate) Functions**

Aggregate functions return a **single result** based on multiple rows.

**2.1 SUM() – Returns the total sum of a column**

Syntax:

**SELECT SUM(column\_name) FROM table\_name;**

Example:

SELECT SUM(sal) FROM emp;

Returns total salary of all employees.

**2.2 AVG() – Returns the average value**

Syntax:

**SELECT AVG(column\_name) FROM table\_name;**

Example:

SELECT AVG(sal) FROM emp;

Returns average salary of all employees.

**2.3 COUNT() – Counts number of rows**

Syntax**:**

**SELECT COUNT(\*) FROM table\_name;**

Example**:**

SELECT COUNT(\*) FROM emp;

Returns number of employees.

**2.4 MIN() and MAX() – Find smallest & largest values**

Syntax**:**

**SELECT MIN(column\_name), MAX(column\_name) FROM table\_name;**

Example**:**

SELECT MIN(sal), MAX(sal) FROM emp;

Returns **minimum and maximum salary** in the table.

**📌 GROUP BY Examples**

-- 1. Maximum salary in the company

SELECT MAX(sal) FROM emp;

-- 2. Maximum salary per department

SELECT deptno, MAX(sal) FROM emp GROUP BY deptno;

-- 3. Employee count per department

SELECT deptno, COUNT(deptno) FROM emp GROUP BY deptno;

-- 4. Total salary & employee count per department

SELECT deptno, SUM(sal), COUNT(deptno) FROM emp GROUP BY deptno;

**📌 HAVING Clause Example**

-- Departments having more than 4 employees

SELECT deptno, COUNT(deptno)

FROM emp

GROUP BY deptno

HAVING COUNT(deptno) > 4;

📌**Departments with total salary above 100000**

select job ,sum(sal) from emp group by job having sum(sal)>5000;

📌 Query: Max Salary in Departments Where Name Contains 'A'

SELECT deptno, MAX(sal)

FROM emp

WHERE ename LIKE '%a%'

GROUP BY deptno;

**📌 Can Single-Row Functions Be Used in Multi-Row Queries?**

🔹 **Example: Using ROUND() (single-row) inside SUM() (multi-row)**

SELECT deptno, ROUND(AVG(sal), 2) FROM emp GROUP BY deptno;

**📌 Multi-Row Functions Ignore NULL Values**

✅ **Aggregate functions like COUNT(), SUM(), AVG() ignore NULL values.**  
🔹 **Example:**

SELECT COUNT(comm) FROM emp; -- Only counts non-null values in comm column

SELECT SUM(sal) FROM emp; -- Ignores NULL salaries

**📌 Errors When Using Aggregate Functions Improperly**

❌ **Error: Mixing aggregate & non-aggregate columns without GROUP BY**

SELECT SUM(sal), sal FROM emp;

-- ❌ ERROR: 'sal' must be in GROUP BY if SUM() is used.

✅ **Fix: Use GROUP BY**

SELECT deptno, SUM(sal) FROM emp GROUP BY deptno;

❌ **Error: Using Multi-Row Function in WHERE Clause**

SELECT ename FROM emp WHERE sal = MAX(sal);

-- ❌ ERROR: WHERE cannot use multi-row functions like MAX()

✅ **Fix: Use a subquery**

SELECT ename FROM emp WHERE sal = (SELECT MAX(sal) FROM emp);

# **Points to Remember (GROUP BY, HAVING & Aggregate Functions)**

## **CORE CONCEPT POINTS**

1. **Aggregate (Multi-row) functions** work on **multiple rows** and return **a single value**.
2. Common aggregate functions are:
   * COUNT()
   * SUM()
   * AVG()
   * MIN()
   * MAX()
3. **Aggregate functions ignore NULL values**.
   * **Exception:** COUNT(\*) includes NULL values.
4. **GROUP BY** is used to **group rows that have the same values** in specified columns
5. If **GROUP BY** is used, **every column in the SELECT statement must be**:

* Either present in the GROUP BY clause
* Or used inside an **aggregate function**

1. Selecting a column that is **not grouped or non aggregated** will cause an error.

**WHERE & HAVING**

1. **WHERE clause filters individual rows** before grouping is applied.
2. **HAVING clause filters grouped results** after grouping is applied.
3. **WHERE cannot be used with aggregate functions**, but **HAVING can**.

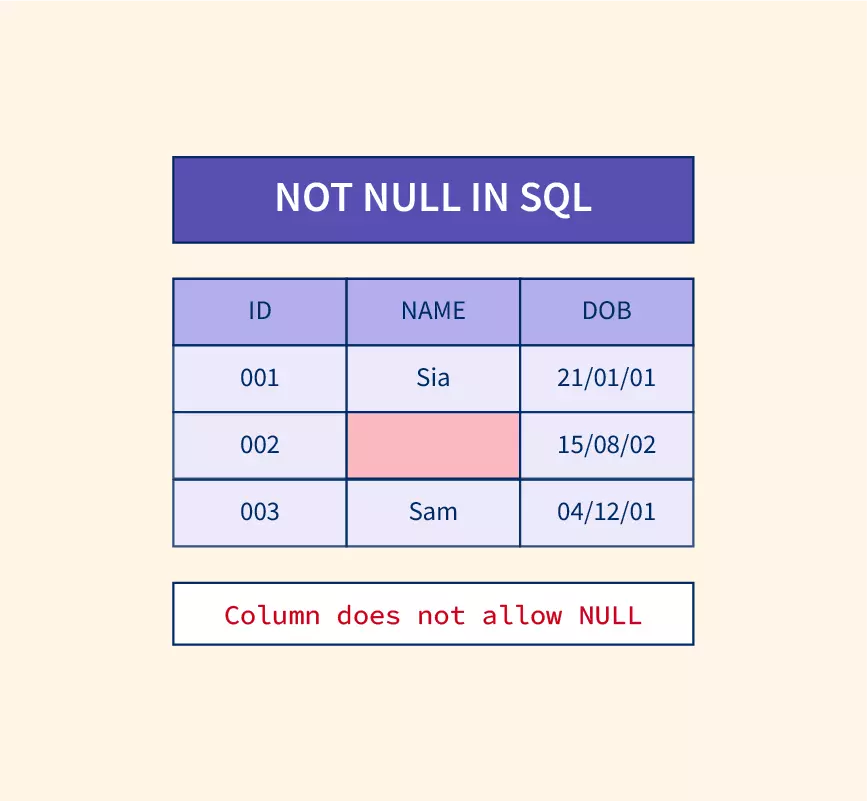
**COUNT() IMPORTANT BEHAVIOR**

1. COUNT(\*)
   * Counts **all rows**
   * Includes rows containing **NULL values**
2. COUNT(column\_name)
   * Counts **only non-NULL values**
   * Ignores NULL values

Constraints

Constraints are the rules applied to table columns to enforce data integrity.

**NOT NULL**

****

* **Purpose**: Ensures that a column cannot have a **NULL** value.
* **Ways to use**: While creating or modifying a table.
* Syntax: Applied at column definition.

1st

CREATE TABLE employees (

id INT **NOT NULL**,

name VARCHAR(100) **NOT NULL**

);

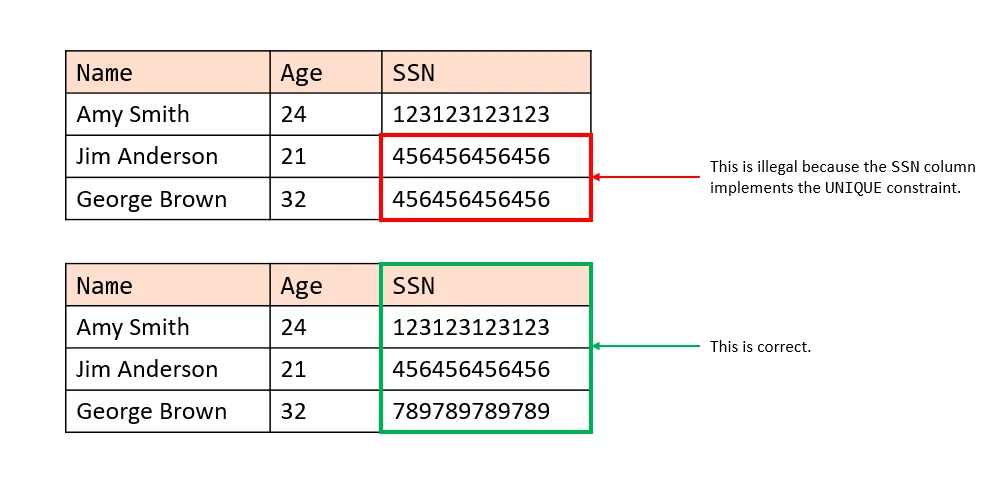
2nd

ALTER TABLE employees MODIFY id INT NOT NULL;

INSERT INTO employees (name) VALUES ('John'); -- ❌ Error: id cannot be NULL

**UNIQUE**

* **Purpose**: create table std (id int,name varchar(20), passportid int unique);
* **Ways to use**: During table creation or using ALTER TABLE.



**1st**

CREATE TABLE students (

id INT UNIQUE,

name VARCHAR(100)

);

**2nd**

CREATE TABLE students (

id INT,

name VARCHAR(100),

phone\_no INT,

UNIQUE(id)

);

**3rd**

ALTER TABLE students ADD UNIQUE(phone\_no);

**🔻 Dropping UNIQUE constraint**

**ALTER TABLE students DROP INDEX phone\_no;**

Ex

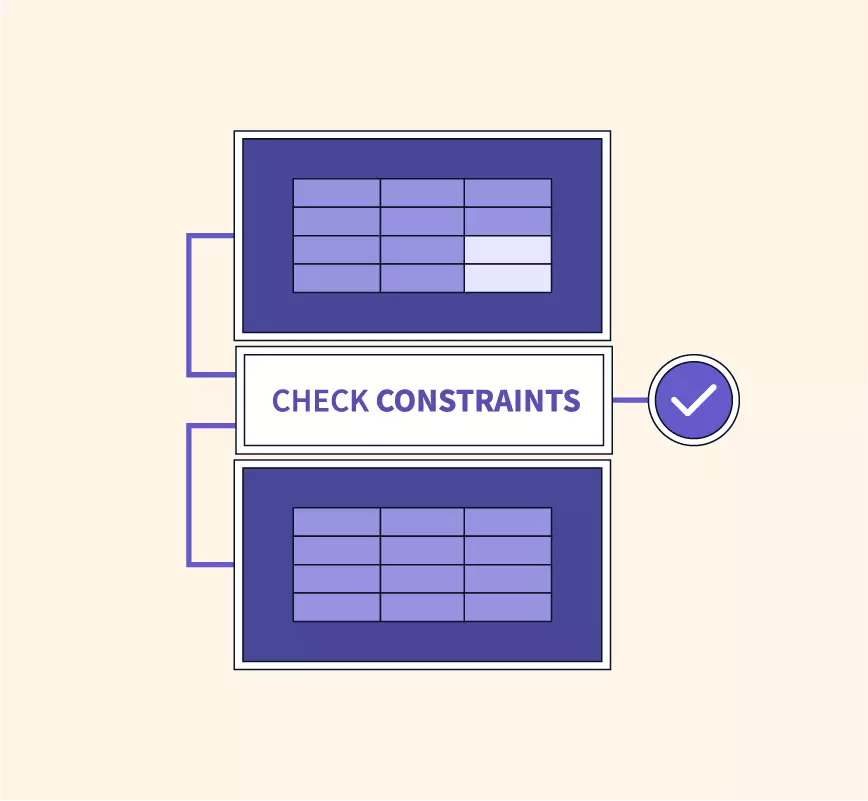
INSERT INTO students (id, name) VALUES (10,”john”);

INSERT INTO students (id, name) VALUES (10,”john”); -- ❌ Error: Duplicate entry

INSERT INTO students (name) VALUES ('Raja'); -- ✅ Allowed: NULL in UNIQUE column

**CHECK**

* **Purpose**: Ensures that the values in a column meet a specified condition.



* **Note**: As of MySQL 8.0.16, **CHECK** constraints are parsed but ignored, as they are not enforced.
* **Ways to use**

While creating a table.

Using ALTER TABLE

CREATE TABLE voters (

name VARCHAR(100),

age INT CHECK (age >= 18)

);

CREATE TABLE voters (

name VARCHAR(100),

age INT,

CONSTRAINT chk\_age CHECK (age >= 18)

);

**Adding CHECK to an existing table**

ALTER TABLE voters ADD CONSTRAINT chk\_name CHECK (LENGTH(name) > 8);

**🔻 Dropping CHECK**

ALTER TABLE voters DROP CHECK chk\_age;

**Ex-**

INSERT INTO voters (name, age) VALUES ('John', 17); -- ❌ Error: Age must be >= 18

**DEFAULT**

* **Purpose**: Sets a default value for a column if no value is specified.
* **Ways to use**

During table creation.

Using ALTER TABLE to modify an existing column.

CREATE TABLE students (

id INT NOT NULL DEFAULT 45,

name VARCHAR(100)

);

**Ex-**

INSERT INTO students (name) VALUES (john); -- ✅ ID will be 45

INSERT INTO students (id, name) VALUES (1,jane); -- ✅ ID = 1

INSERT INTO students (name) VALUES (‘JESSY’); -- ✅ ID will be 45

**🔻 Modifying DEFAULT**

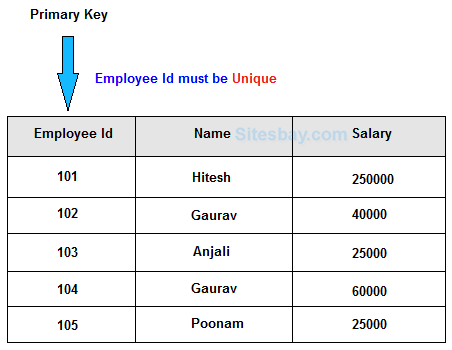
ALTER TABLE students MODIFY id INT DEFAULT 50;

**🔻 Dropping DEFAULT**

ALTER TABLE students ALTER COLUMN id DROP DEFAULT;

**5. PRIMARY KEY**

* **Purpose**: Uniquely identifies each row in a table.



* **Note**: Combines NOT NULL and UNIQUE constraints.
* Ways to use:

While creating a table.

Using ALTER TABLE.

Dropping a primary key.

CREATE TABLE employees (

id INT PRIMARY KEY,

name VARCHAR(100) NOT NULL

);

CREATE TABLE employees (

id INT,

name VARCHAR(100),

PRIMARY KEY(id)

);

ALTER TABLE employees ADD PRIMARY KEY (id);

**🔻 Dropping PRIMARY KEY**

ALTER TABLE employees DROP PRIMARY KEY;

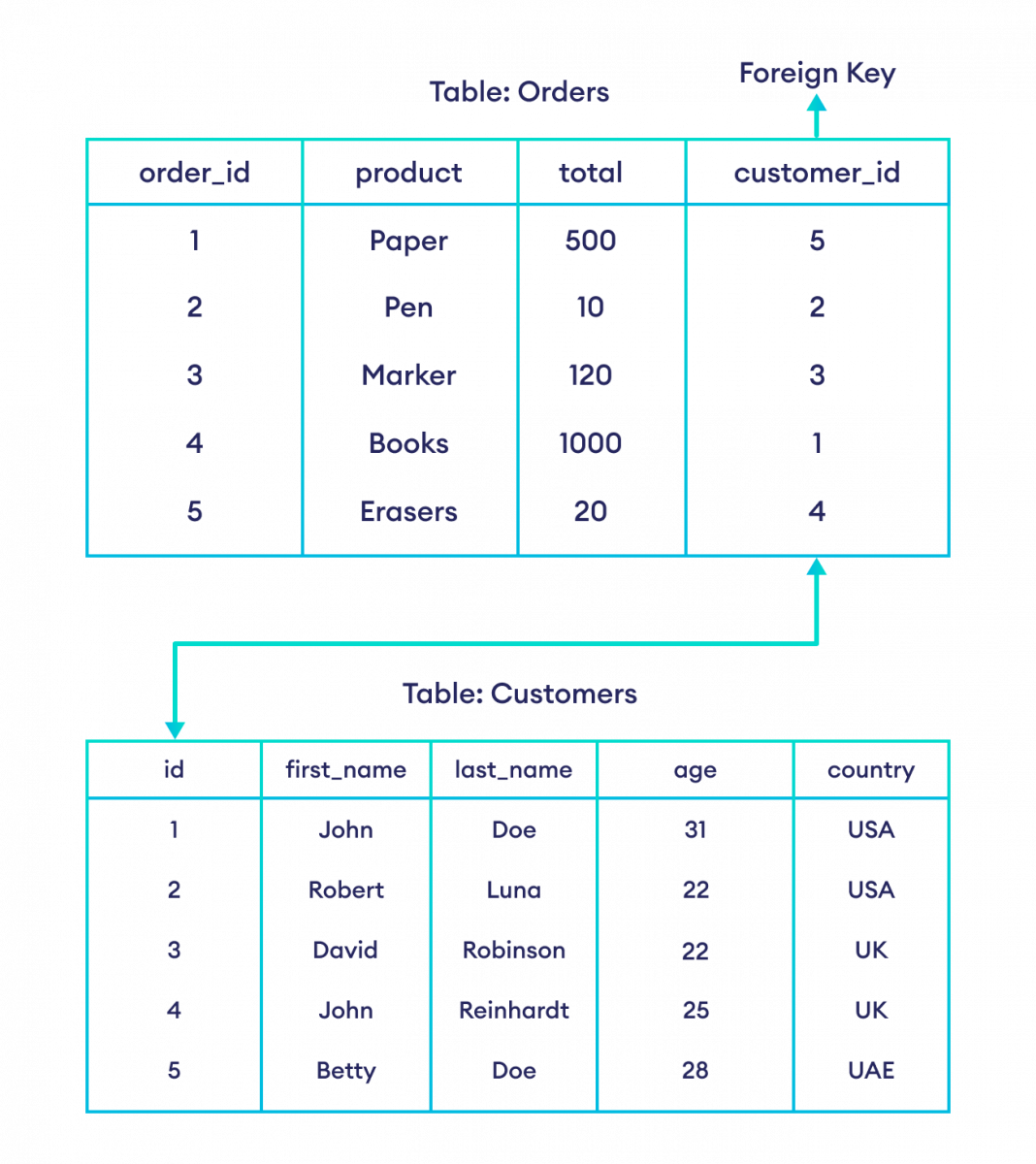
**Ex-**

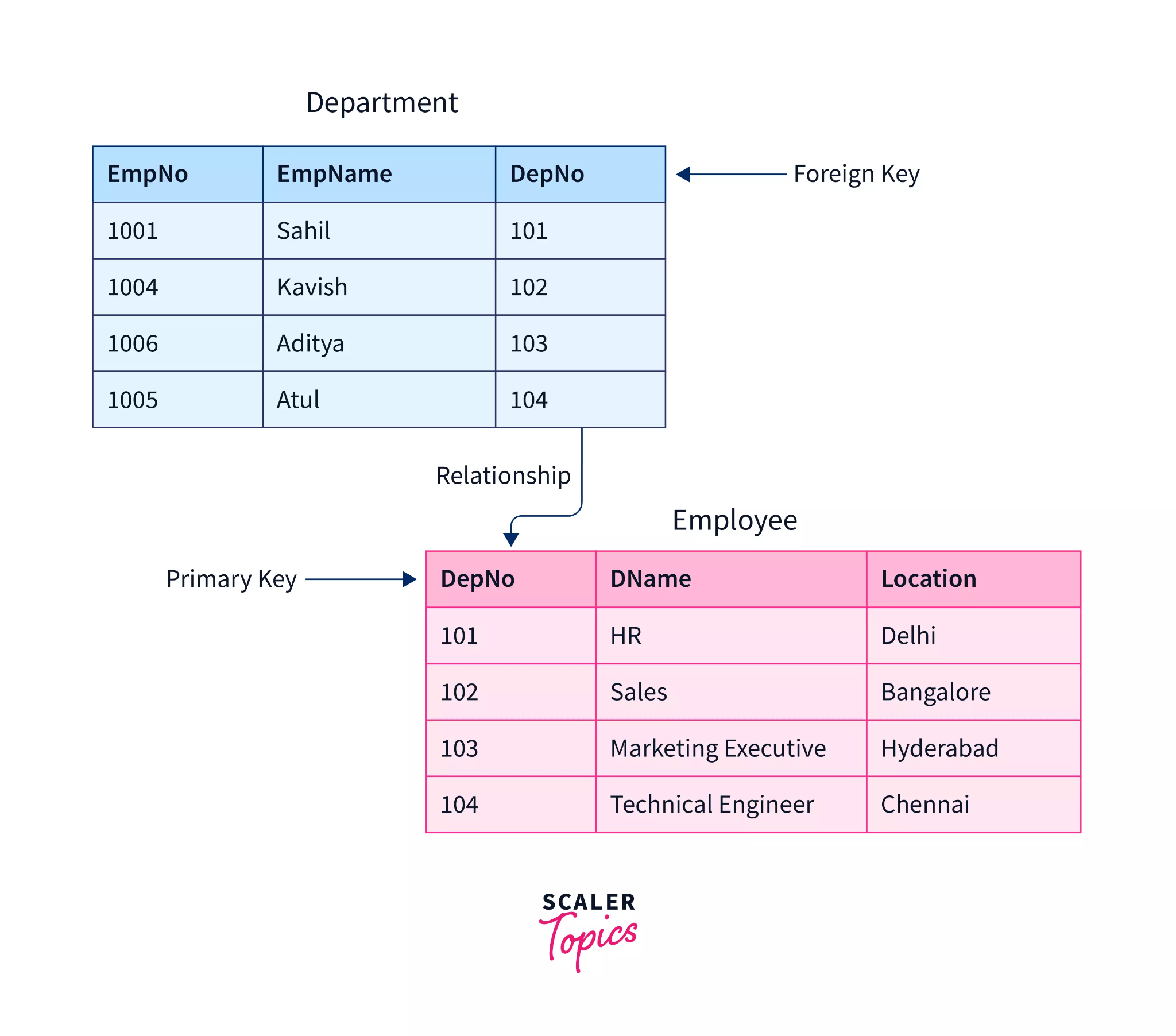
INSERT INTO employees (id, name) VALUES (1, 'John');

INSERT INTO employees (id, name) VALUES (1, 'Jane'); -- ❌ Error: Duplicate id

**6. FOREIGN KEY**

* **Purpose**: Ensures referential integrity by linking two tables.
* **Note**: The referenced table should have a PRIMARY KEY.
* Foreign Key column can have duplicates and NULL values.
* Parent table: The table with a PRIMARY KEY. and foreign key
* Child table: The table referencing the PRIMARY KEY.

****

**create table dept (empno int primary key, ename varchar(20), deptno int, foreign key(deptno) references emp(deptno));**

**Creating Foreign Key**

CREATE TABLE departments (

dept\_id INT PRIMARY KEY,

dept\_name VARCHAR(100) NOT NULL

);

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(100) NOT NULL,

dept\_id INT,

FOREIGN KEY (dept\_id) REFERENCES departments(dept\_id)

);

**🔻 Dropping FOREIGN KEY**

ALTER TABLE employees DROP FOREIGN KEY dept\_id;

Ex-

INSERT INTO employees (emp\_id, emp\_name, dept\_id) VALUES (1, 'Alice', 10); -- ❌ Error: No department with id 10

**Ways to add a Foreign Key**

**-- Without named constraint**

**CREATE TABLE employees (**

**emp\_id INT,**

**dept\_id INT,**

**FOREIGN KEY (dept\_id) REFERENCES departments(dept\_id)**

**);**

**-- With named constraint**

**CREATE TABLE employees (**

**emp\_id INT,**

**dept\_id INT,**

**CONSTRAINT fk\_dept FOREIGN KEY (dept\_id) REFERENCES departments(dept\_id)**

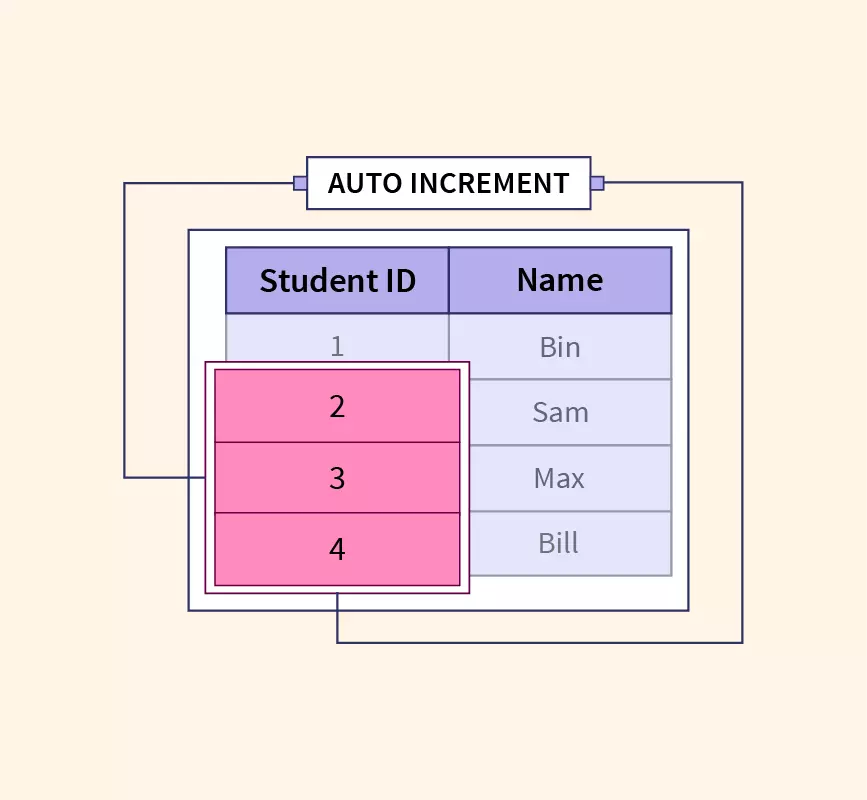
**);**

**-- Adding to an existing table**

**ALTER TABLE employees ADD CONSTRAINT fk\_dept FOREIGN KEY (dept\_id) REFERENCES departments(dept\_id);**

**7. AUTO\_INCREMENT**

* **Purpose**: Automatically generates a **unique** value for each row.



* **Note**: Only one AUTO\_INCREMENT column is allowed per table.
* **Works only with**: PRIMARY KEY.
* **By default, starts at 1** but can be customized.

**CREATE TABLE students (**

**id INT PRIMARY KEY AUTO\_INCREMENT,**

**name VARCHAR(100)**

**);**

**Ex-**

INSERT INTO students (name) VALUES ('Jenny'); -- ID = 1

INSERT INTO students (name) VALUES ('Jack'); -- ID = 2

🔻 **Changing AUTO\_INCREMENT start value**

**ALTER TABLE students AUTO\_INCREMENT = 10;**

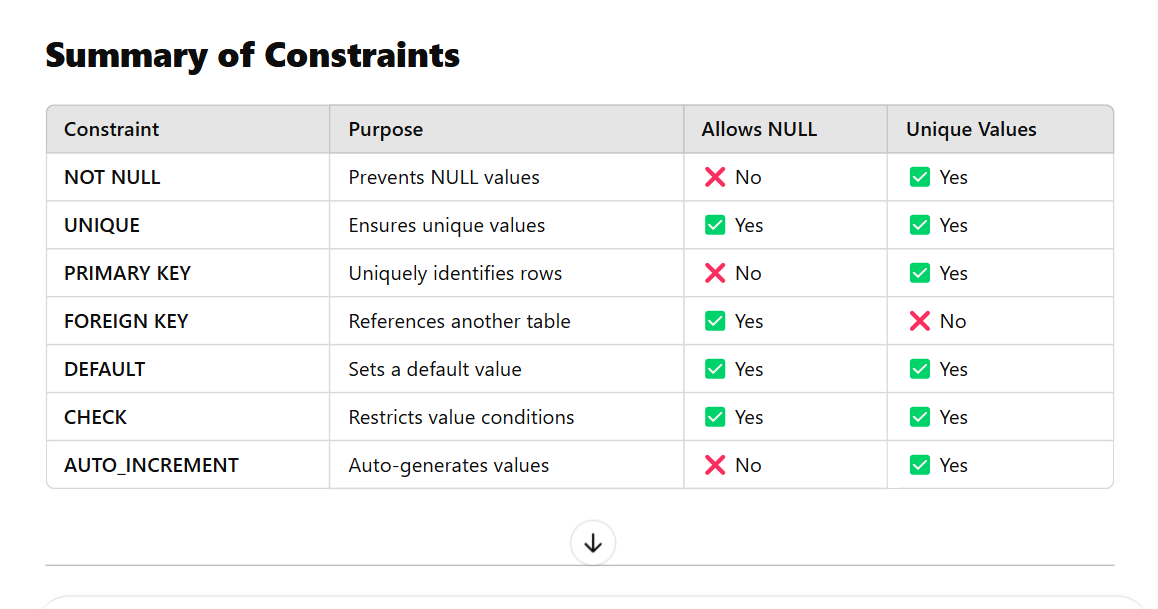
🔻 **Changing Increment Step**

**SET @@auto\_increment\_increment = 5;**

Ex-

INSERT INTO students (name) VALUES ('Jane'); -- ID = 10

INSERT INTO students (name) VALUES ('Mike'); -- ID = 15



## **MySQL Constraints -Use Cases**

### **🔒 UNIQUE**

| **Use Case** | **Real-World Example** |
| --- | --- |
| Email uniqueness | One email per user |
| Username uniqueness | Login systems |
| Phone number | Telecom customers |
| National ID | Government records |
| Card number | Banking cards |

### **❗ NOT NULL**

| **Use Case** | **Real-World Example** |
| --- | --- |
| User credentials | Username & password |
| Order details | Order date required |
| Payroll data | Salary field |
| Medical data | Diagnosis |
| HR data | Joining date |

### **PRIMARY KEY**

| **Use Case** | **Real-World Example** |
| --- | --- |
| User identification | user\_id in user table |
| Order tracking | order\_id in orders |
| Employee records | employee\_id in HR |
| Patient records | patient\_id in hospital |
| Bank accounts | account\_id in banking |

### **✔️ CHECK**

| **Use Case** | **Real-World Example** |
| --- | --- |
| Age validation | Age ≥ 18 |
| Salary validation | Salary > 0 |
| Exam scores | Marks 0–100 |
| Product pricing | Price ≥ cost |
| Status validation | ACTIVE / INACTIVE |

### **🧩 DEFAULT**

| **Use Case** | **Real-World Example** |
| --- | --- |
| Account status | Default = ACTIVE |
| Order creation date | Current date |
| Inventory quantity | Default = 0 |
| Notification state | Unread by default |
| Audit timestamp | Auto created time |

### **🔗 FOREIGN KEY**

| **Use Case** | **Real-World Example** |
| --- | --- |
| Customer → Orders | Orders linked to customers |
| Account → Transactions | Transactions linked to accounts |
| Department → Employees | Employees in valid departments |
| Course → Enrollments | Student enrollments |
| Doctor → Appointments | Medical appointments |

MySQL Joins

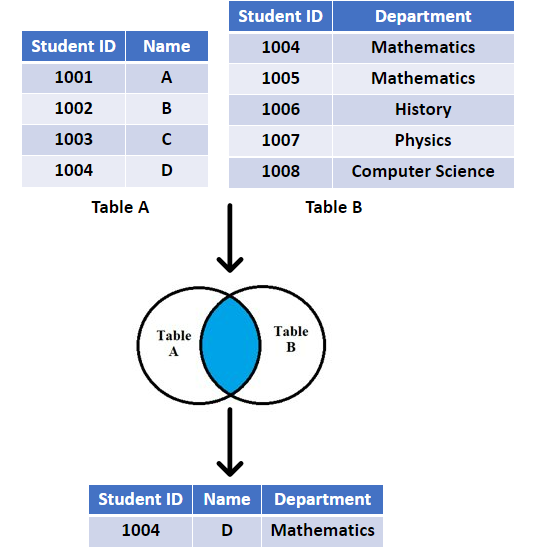
**Joins in MySQL**

Joins in MySQL are used to retrieve data from multiple tables based on a related column. The different types of joins available are:



**1. INNER JOIN**

**Explanation:**

* The INNER JOIN keyword selects records that have matching values in both tables.
* If there is no match, the rows are excluded from the result set.
* This is the most commonly used join in SQL.
* It is useful when we only need records that have corresponding entries in both tables.
* 
* Syntax:

**SELECT columns FROM table1**

**INNER JOIN table2**

**ON table1.column = table2.column;**

* Example Queries:

SELECT \* FROM emp

INNER JOIN dept

ON emp.deptno = dept.deptno;

SELECT empno, ename FROM emp

INNER JOIN dept

ON emp.deptno = dept.deptno;

-- This query will throw an error due to ambiguity as deptno exists in both tables:

SELECT empno, deptno FROM emp

INNER JOIN dept

ON emp.deptno = dept.deptno;

* -- Correcting ambiguity by specifying the table name or alias:

SELECT empno, emp.deptno FROM emp

INNER JOIN dept

ON emp.deptno = dept.deptno;

SELECT empno, e.deptno FROM emp e

INNER JOIN dept d

ON e.deptno = d.deptno;

**Joins dummy data**

-- Create the Database

CREATE DATABASE IF NOT EXISTS joinsdata;

USE joinsdata;

-- 1. Create and Insert into Students Table (Added 'city' column)

CREATE TABLE Students (

student\_id INT PRIMARY KEY,

student\_name VARCHAR(50),

email VARCHAR(100),

city VARCHAR(50)

);

INSERT INTO Students (student\_id, student\_name, email, city) VALUES

(1, 'Ramesh', 'ramesh@gmail.com', 'Delhi'),

(2, 'Sita', 'sita@gmail.com', 'Mumbai'),

(3, 'Arjun', 'arjun@gmail.com', 'Bangalore'),

(4, 'Meena', 'meena@gmail.com', 'Chennai');

-- 2. Create and Insert into Courses Table

CREATE TABLE Courses (

course\_id INT PRIMARY KEY,

course\_name VARCHAR(50),

fee DECIMAL(10, 2)

);

INSERT INTO Courses (course\_id, course\_name, fee) VALUES

(101, 'Python', 15000.00),

(102, 'Java', 18000.00),

(103, 'SQL', 10000.00),

(104, 'React', 20000.00);

-- 3. Create and Insert into Trainers Table

CREATE TABLE Trainers (

trainer\_id INT PRIMARY KEY,

trainer\_name VARCHAR(50),

expertise VARCHAR(50)

);

INSERT INTO Trainers (trainer\_id, trainer\_name, expertise) VALUES

(201, 'Rahul', 'Python'),

(202, 'Anita', 'Java'),

(203, 'Kiran', 'SQL'),

(204, 'Neha', 'React');

-- 4. Create and Insert into Enrollments Table

CREATE TABLE Enrollments (

enroll\_id INT PRIMARY KEY,

student\_id INT,

course\_id INT,

trainer\_id INT,

enroll\_date DATE,

FOREIGN KEY (student\_id) REFERENCES Students(student\_id),

FOREIGN KEY (course\_id) REFERENCES Courses(course\_id),

FOREIGN KEY (trainer\_id) REFERENCES Trainers(trainer\_id)

);

-- Note: In the image, there is a duplicate entry for enroll\_id 5.

-- Valid SQL requires unique Primary Keys, so I have changed the last ID to 6.

INSERT INTO Enrollments (enroll\_id, student\_id, course\_id, trainer\_id, enroll\_date) VALUES

(1, 1, 101, 201, '2024-01-10'),

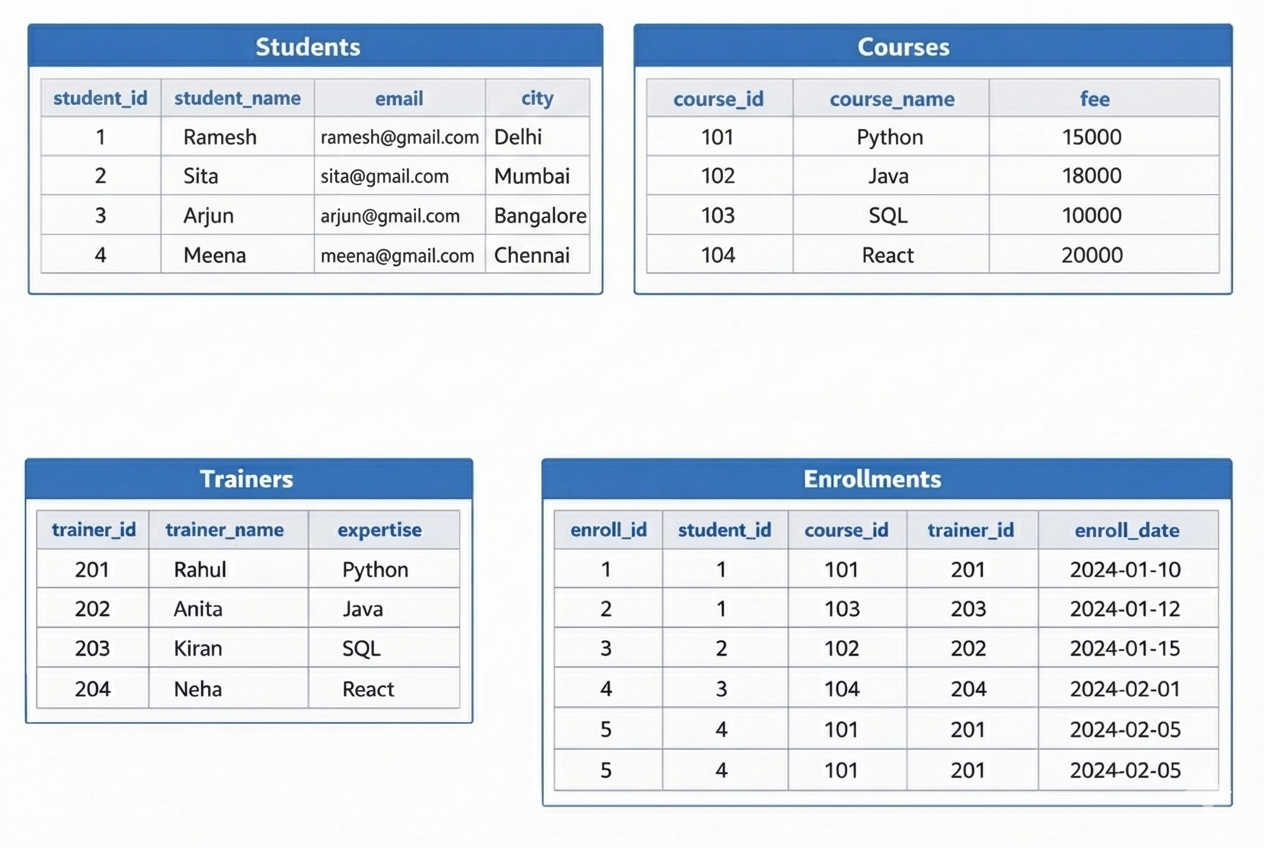
(2, 1, 103, 203, '2024-01-12'),

(3, 2, 102, 202, '2024-01-15'),

(4, 3, 104, 204, '2024-02-01'),

(5, 4, 101, 201, '2024-02-05'),

(6, 4, 101, 201, '2024-02-05');



**Queries on joins**

**1) Display student\_id who registered under anitha**

**2) Display student\_id who registered for python**

**3) Display who was teaching the course having highest fees**

**4) Display sid,cid,cname,fees of each std**

**5) Display studentid,total fees paid by each student**

**6) Display coursename,total fees paid for each course**

**7) Display sid,trianername, subject,fee for all students**

**Advanced**

**1) find the student names who have taken python**

**2)find the count of no of students in each course**

**3)find the students names who are training under anitha**

**4)find the trainers who has maximum enrolled students**

**5)find the total fees of every trainer based on total student under his course**

**6) Find courses with more than 1 student**

**7) Find students who enrolled in more than one course**

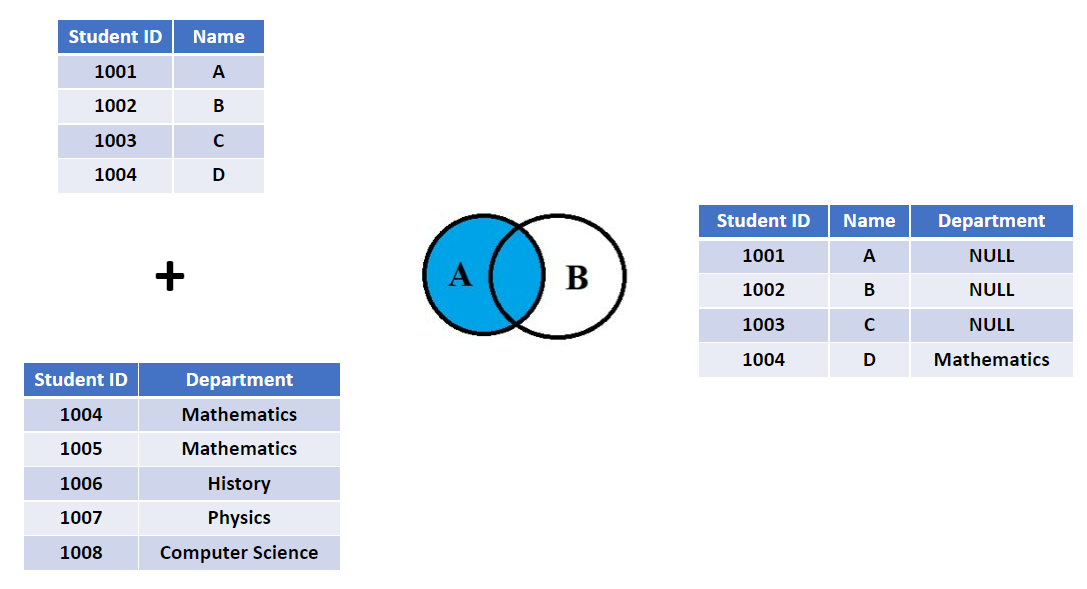
**8) Find trainers who collected fees more than 20,000**

**9) Find students who paid more than 20,000**

**10) Find courses whose total collection is more than 30,000**

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

**Explanation:**

* The LEFT JOIN returns all records from the left table and the matched records from the right table.
* If there is no match, NULL values will be returned for the columns from the right table.
* It ensures that all records from the left table appear in the result set.
* 
* Syntax:

**SELECT \* FROM table1**

**LEFT JOIN table2**

**ON table1.column = table2.column;**

* Example:

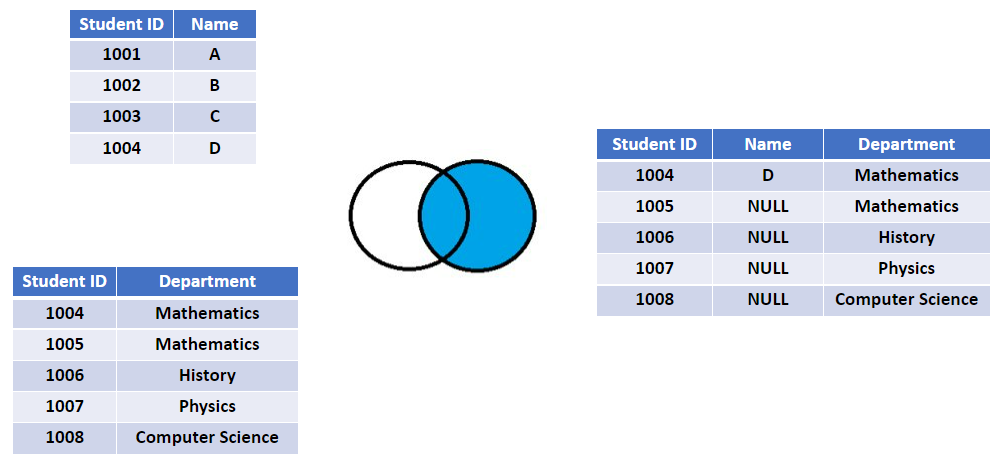
SELECT \* FROM dept

LEFT JOIN emp

ON dept.deptno = emp.deptno;

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

**Explanation:**

* The RIGHT JOIN returns all records from the right table and the matched records from the left table.
* If there is no match, NULL values will be returned for the columns from the left table.
* It is similar to LEFT JOIN, but in reverse.
* 
* Syntax:

**SELECT \* FROM table1**

**RIGHT JOIN table2**

**ON table1.column = table2.column;**

* Example:

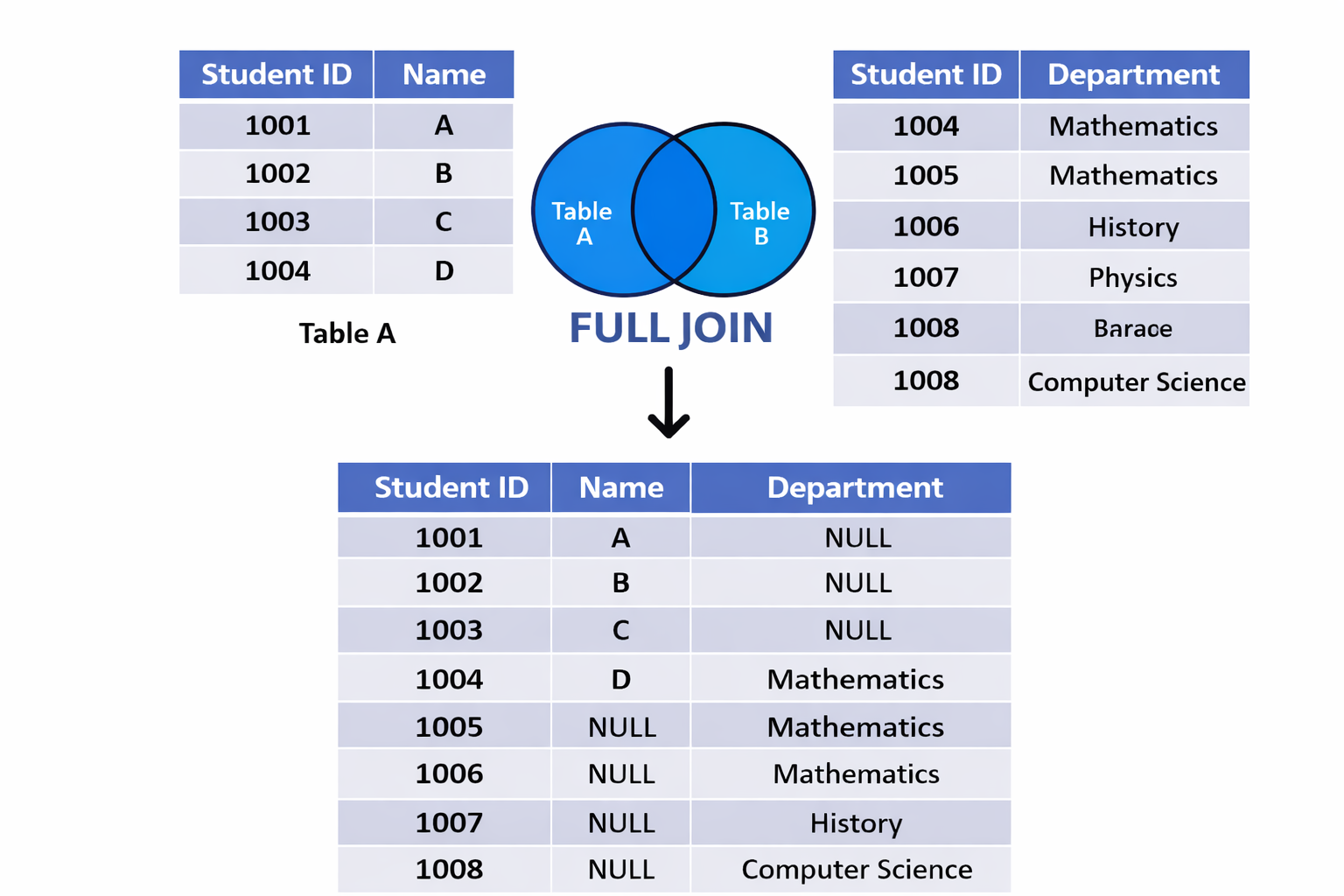
SELECT \* FROM emp

RIGHT JOIN dept

ON emp.deptno = dept.deptno;

**4. FULL OUTER JOIN (Simulated using UNION)**

* Returns all rows when there is a match in either table.
* MySQL does not support FULL OUTER JOIN directly, so we use a combination of LEFT JOIN and RIGHT JOIN with UNION.
* This type of join returns all rows from both tables, with NULL where there is no match.



Example:

**SELECT \* FROM emp**

**LEFT JOIN dept**

**ON emp.deptno = dept.deptno**

**UNION**

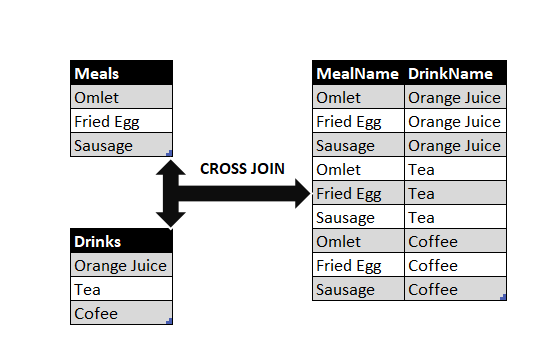
**SELECT \* FROM emp**

**RIGHT JOIN dept**

**ON emp.deptno = dept.deptno;**

**5. CROSS JOIN (Cartesian Join)**

**Explanation:**

* The CROSS JOIN returns the Cartesian product of the two tables.
* Every row from the first table is combined with every row from the second table.
* If table1 has m rows and table2 has n rows, the result set will have m \* n rows.
* It is useful for generating test data but rarely used in real-world applications.
* 
* Syntax:

**SELECT \* FROM table1**

**CROSS JOIN table2;**

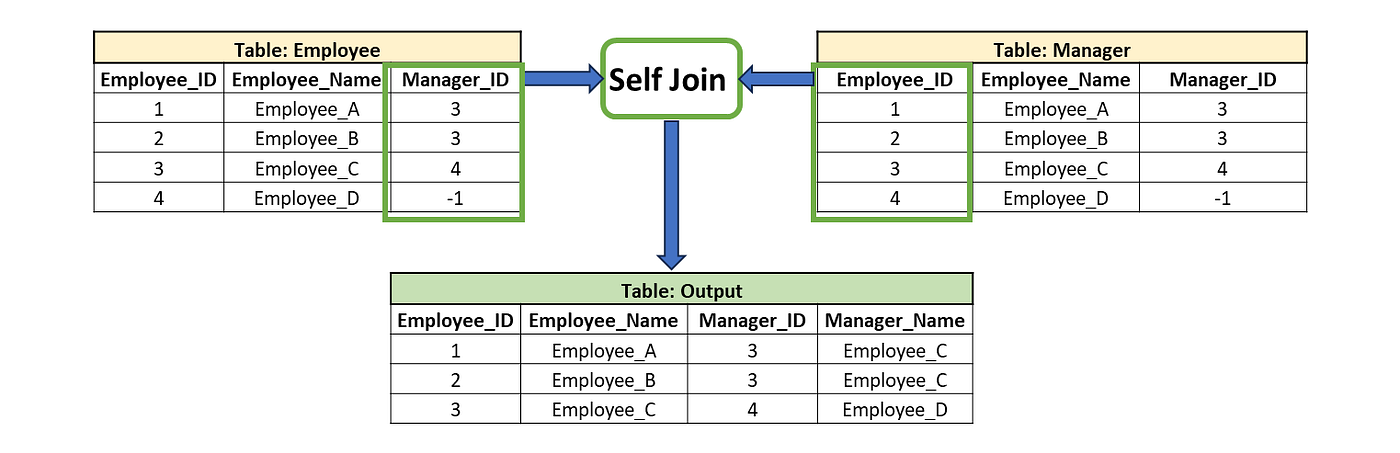
* Example:

SELECT \* FROM emp

CROSS JOIN dept;

**6. SELF JOIN**

**Explanation:**

* A SELF JOIN is when a table is joined with itself.
* This is useful when comparing rows within the same table, such as finding employees and their managers.
* Table aliases are necessary to distinguish between the two instances of the same table.
* It can be an INNER JOIN, LEFT JOIN, or RIGHT JOIN.
* 
* Example Queries:
* -- INNER JOIN Self Join Example:

SELECT \* FROM emp e1

INNER JOIN emp e2

ON e1.mgr = e2.empno;

SELECT e1.empno, e1.ename, e1.mgr, e2.ename

FROM emp e1

INNER JOIN emp e2

ON e1.mgr = e2.empno;

* -- RIGHT JOIN Self Join Example:

SELECT e1.empno, e1.ename, e1.mgr, e2.ename

FROM emp e1

RIGHT JOIN emp e2

ON e1.mgr = e2.empno;

* -- LEFT JOIN Self Join Example:

SELECT e1.empno, e1.ename, e1.mgr, e2.ename

FROM emp e1

LEFT JOIN emp e2

ON e1.mgr = e2.empno;

**Types of joins**

**DROP DATABASE IF EXISTS types;**

**CREATE DATABASE types;**

**USE types;**

**CREATE TABLE Students (**

**student\_id INT PRIMARY KEY,**

**student\_name VARCHAR(50),**

**email VARCHAR(100),**

**city VARCHAR(50)**

**);**

**INSERT INTO Students VALUES**

**(1, 'Ramesh', 'ramesh@gmail.com', 'Delhi'),**

**(2, 'Sita', 'sita@gmail.com', 'Mumbai'),**

**(3, 'Arjun', 'arjun@gmail.com', 'Bangalore'),**

**(4, 'Meena', 'meena@gmail.com', 'Chennai'),**

**(5, 'Kumar', 'kumar@gmail.com', 'Hyderabad'),**

**(6, 'Anjali', 'anjali@gmail.com', 'Pune');**

**CREATE TABLE Courses (**

**course\_id INT PRIMARY KEY,**

**course\_name VARCHAR(50),**

**fee DECIMAL(10,2)**

**);**

**INSERT INTO Courses VALUES**

**(101, 'Python', 15000),**

**(102, 'Java', 18000),**

**(103, 'SQL', 10000),**

**(104, 'React', 20000),**

**(105, 'Django', 22000),**

**(106, 'NodeJS', 19000);**

**CREATE TABLE Trainers (**

**trainer\_id INT PRIMARY KEY,**

**trainer\_name VARCHAR(50),**

**expertise VARCHAR(50)**

**);**

**INSERT INTO Trainers VALUES**

**(201, 'Rahul', 'Python'),**

**(202, 'Anita', 'Java'),**

**(203, 'Kiran', 'SQL'),**

**(204, 'Neha', 'React'),**

**(205, 'Suresh', 'Django'),**

**(206, 'Amit', 'NodeJS');**

**CREATE TABLE Enrollments (**

**enroll\_id INT PRIMARY KEY,**

**student\_id INT,**

**course\_id INT,**

**trainer\_id INT,**

**enroll\_date DATE**

**);**

**INSERT INTO Enrollments VALUES**

**(1, 1, 101, 201, '2024-01-10'),**

**(2, 1, 103, 203, '2024-01-12'),**

**(3, 2, 102, 202, '2024-01-15'),**

**(4, 3, 104, 204, '2024-02-01'),**

**(5, 4, 101, 201, '2024-02-05'),**

**(6, 4, NULL, 202, '2024-02-10'), -- Missing course**

**(7, 3, 103, NULL, '2024-02-12'); -- Missing trainer**

**ALTER TABLE Students**

**ADD mentor\_id INT;**

**UPDATE Students SET mentor\_id = 1 WHERE student\_id IN (2,3);**

**UPDATE Students SET mentor\_id = 2 WHERE student\_id = 4;**

**UPDATE Students SET mentor\_id = 3 WHERE student\_id = 6;**

****

**inner join**

**1) Students with enrolled courses**

**LEFT JOIN**

**2) All students + enrollments**

**3) Students with NO enrollment**

**RIGHT JOIN**

**4) erollments +All courses**

**5) Courses with NO students**

**FULL JOIN**

**6) All Trainers +All Enrollments**

**SELF JOIN**

**7) Student with their Mentor Name**

**8) Students who doesn’t have a mentor**

**9) Mentor with Number of Students**

**CROSS JOIN**

**10) Get employees whose salary is greater than FORD’s salary**

**INNER JOIN**

**Students with enrolled courses**

SELECT s.student\_name, c.course\_name

FROM Students s

INNER JOIN Enrollments e ON s.student\_id = e.student\_id

INNER JOIN Courses c ON e.course\_id = c.course\_id;

**LEFT JOIN**

**All students + enrollments ids**

SELECT s.student\_name, e.enroll\_id

FROM Students s

LEFT JOIN Enrollments e ON s.student\_id = e.student\_id;

**Students with NO enrollment**

SELECT s.student\_name

FROM Students s

LEFT JOIN Enrollments e ON s.student\_id = e.student\_id

WHERE e.enroll\_id IS NULL;

**Students + courses (even if missing)**

SELECT s.student\_name, c.course\_name

FROM Students s

LEFT JOIN Enrollments e ON s.student\_id = e.student\_id

LEFT JOIN Courses c ON e.course\_id = c.course\_id;

**RIGHT JOIN**

**All courses + enrollments**

SELECT c.course\_name, e.enroll\_id

FROM Enrollments e

RIGHT JOIN Courses c ON e.course\_id = c.course\_id;

**Courses with NO students**

SELECT c.course\_name

FROM Enrollments e

RIGHT JOIN Courses c ON e.course\_id = c.course\_id

WHERE e.enroll\_id IS NULL;

**Courses + students**

SELECT c.course\_name, s.student\_name

FROM Enrollments e

RIGHT JOIN Courses c ON e.course\_id = c.course\_id

LEFT JOIN Students s ON e.student\_id = s.student\_id;

**FULL JOIN**

### **Trainers + Enrollments**

SELECT t.trainer\_name, e.enroll\_id

FROM Trainers t

LEFT JOIN Enrollments e ON t.trainer\_id = e.trainer\_id

UNION

SELECT t.trainer\_name, e.enroll\_id

FROM Trainers t

RIGHT JOIN Enrollments e ON t.trainer\_id = e.trainer\_id;

**SELF JOIN**

**Student with their Mentor Name**

SELECT

s.student\_name AS student,

m.student\_name AS mentor

FROM Students s

JOIN Students m

ON s.mentor\_id = m.student\_id;

**Students who doesn’t have a mentor**

select \* from students s1 left join students s2 on s1.mentor\_id=s2.student\_id where s2.student\_id is null;

**Mentor with Number of Students**

SELECT

m.student\_name AS mentor,

COUNT(s.student\_id) AS total\_students

FROM Students s

JOIN Students m

ON s.mentor\_id = m.student\_id

GROUP BY m.student\_name;

**CROSS JOIN**

**Students × Courses**

SELECT s.student\_name, c.course\_name

FROM Students s

CROSS JOIN Courses c;

**Get employees whose salary is greater than FORD’s salary**

**Using CROSS JOIN**

SELECT e.ename, e.sal

FROM emp e

CROSS JOIN emp f

WHERE f.ename = 'FORD'

AND e.sal > f.sal;

MySQL Indexes

**What is an Index in MySQL?**

* An **index** is a mechanism that enables **fast lookup** of rows in a table by creating a reference to their location in the database.
* It allows MySQL to quickly locate data **without scanning the entire table**.
* Indexes **improve query performance**, especially for large datasets.
* They are automatically **maintained** by MySQL during INSERT, UPDATE, and DELETE operations.
* **Avoid over-indexing** as it consumes disk space and memory.

**How Indexes Work?**

* The term **"lookup"** generally refers to querying the database to retrieve specific information.
* This is achieved using the SELECT statement, potentially with conditions specified in the WHERE clause to narrow down the results.
* Indexes store data **alongside table data** for fast access.

**Types of Indexes in MySQL**

**1️ Primary Index (Primary Key)**

* Automatically created when a PRIMARY KEY is defined.
* Ensures **unique identification** of each row.
* Example:

CREATE TABLE customers (

cid INT PRIMARY KEY, -- MySQL creates an index automatically

cname VARCHAR(10)

);

**2️ Foreign Key Index**

* Automatically created for FOREIGN KEY constraints.
* Used to maintain relationships between tables.
* Example:

CREATE TABLE orders (

oid INT PRIMARY KEY,

pname VARCHAR(10),

amount DECIMAL(5,2),

cid INT,

FOREIGN KEY (cid) REFERENCES customers (cid) -- Foreign key index is created

);

**3️ Unique Index**

* Ensures all values in a column are unique.
* Example:

CREATE UNIQUE INDEX indx\_name ON tablename (col1);

CREATE UNIQUE INDEX idx\_uni ON products (pid);

create unique index idx\_pname on orders(pname);

**4️ Regular (Non-Unique) Index**

* Used to speed up searches but allows duplicate values.
* Example:

CREATE INDEX idx\_name ON tablename (colname);

CREATE INDEX idx\_pname ON products (pname);

**5️ Composite Index**

* An index on **multiple columns**.
* Useful for queries filtering by multiple conditions.
* Example:

CREATE INDEX idx\_name ON tablename (column1, column2);

CREATE INDEX idx\_name ON orders (pname, amount);

**6️ Full-Text Index**

* Used for **fast text searches** in large text fields.
* Example:
* CREATE FULLTEXT INDEX idx\_description ON articles(content);

**7️ patial Index**

* Used for **geographic data** (e.g., POINT, LINESTRING, POLYGON).
* Example:
* CREATE SPATIAL INDEX idx\_location ON places(location);

**Creating Indexes in MySQL**

**🔹 Creating an Index at Table Creation**

CREATE TABLE tablename (

col DATATYPE,

col1 DATATYPE,

INDEX (col1)

);

Example:

CREATE TABLE products (

pid INT,

pname VARCHAR(10),

INDEX (pid)

);

**🔹 Adding an Index to an Existing Table**

CREATE INDEX index\_name ON tablename (column1);

Example:

CREATE INDEX idx\_pname ON products (pname);

**🔹 Creating an Index with Primary Key**

CREATE TABLE users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(10)

);

**Dropping (Removing) Indexes**

DROP INDEX indexname ON tablename;

Example:

DROP INDEX idx\_pname ON products;

**Checking Existing Indexes**

SHOW INDEX FROM tablename;

Example:

SHOW INDEX FROM orders;

MySQL Subqueries

**What is a Subquery ?**

**A subquery is a query that is nested inside another query. It is used to retrieve data that will be used in the main query (also called the outer query).**

* A subquery is enclosed in parentheses ().
* It can be placed in SELECT, FROM, or WHERE clauses.
* The result of a subquery is passed to the outer query.
* Subqueries can return single values, multiple values, or even entire tables**.**

**Types of Subqueries**

**1️ single-Row Subquery**

**A single-row subquery returns only one value (one row, one column) and is used with operators like =, >, <, >=, <=.**

**🔹 Example: Find employees who have the same salary as the highest salary in department 10.**

**SELECT empname, salary**

**FROM employees**

**WHERE salary = (SELECT MAX(salary) FROM employees WHERE dept\_id = 10);**

✔ The subquery finds the maximum salary in department 10.  
✔ The outer query retrieves employees with that salary.

**2️ Multi-Row Subquery**

**A multi-row subquery returns multiple rows and is used with operators like IN, ANY, ALL.**

**🔹 Example: Find employees who have the same salary as any employee in department 20.**

**SELECT empname, salary**

**FROM employees**

**WHERE salary IN (SELECT salary FROM employees WHERE dept\_id = 20);**

✔ The subquery returns a list of salaries in department 20.  
✔ The outer query retrieves employees whose salaries match any of those values.

**✅ Using ANY and ALL:**

* ANY: Checks if at least one value from the subquery satisfies the condition.
* ALL: Requires that all values from the subquery satisfy the condition.

**🔹 Example (ANY): Find employees earning more than the lowest salary in department 30.**

**SELECT empname, salary**

**FROM employees**

**WHERE salary > ANY (SELECT salary FROM employees WHERE dept\_id = 30);**

**🔹 Example (ALL): Find employees earning more than all salaries in department 20**

**SELECT empname, salary**

**FROM employees**

**WHERE salary > ALL (SELECT salary FROM employees WHERE dept\_id = 20);**

**🔹Employees earning more than ALL analysts.---analyst column is job**

**SELECT ename, sal**

**FROM emp**

**WHERE sal > ALL (SELECT sal FROM emp WHERE job = 'ANALYST');**

**3️ Subquery in FROM Clause (Derived Table)**

**A subquery can act as a temporary table inside the FROM clause.**

**🔹 Example: Find the highest salary in each department.**

**select deptno,max(sal)**

**from (select deptno, sal from emp) as e**

**group by deptno;**

**4️ Subquery in SELECT Clause**

**A subquery can be used inside the SELECT clause to return a single value per row.**

**🔹 Example: Get employees along with their department’s average salary.**

**SELECT ename, sal, (SELECT AVG(sal) FROM emp WHERE deptno = e.deptno)**

**AS avg\_salary FROM emp e;**

**✔ The subquery calculates the average salary per department.  
✔ The outer query retrieves each employee’s salary alongside their department’s average salary.**

**Performance Considerations**

* **Optimize subqueries when working with large datasets.**
* **Use indexes on columns in subqueries for faster lookups.**
* **If a subquery returns a large number of values, consider using a JOIN instead.**

**🔹 Example (Using JOIN instead of a Subquery):**

**SELECT e.empname, e.salary, d.avg\_salary**

**FROM employees e**

**JOIN (SELECT dept\_id, AVG(salary) AS avg\_salary FROM employees GROUP BY dept\_id) d**

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

**WHERE e.salary > d.avg\_salary;**

**✔ This performs better than a correlated subquery.**

**Key Takeaways**

**✅ Single-row subqueries return one value.  
✅ Multi-row subqueries return multiple values (use IN, ANY, ALL).  
✅ Correlated subqueries depend on the outer query’s data.  
✅ Derived tables (subqueries in FROM) act as temporary tables.  
✅ Use JOINs for better performance when possible.**

**Subqueries**

1. **Employees who work in the Sales department**
2. **Employees earning more than the average salary**
3. **Employees who work in the same department as SMITH**
4. **Highest paid employee**
5. **Employees who work in NEW YORK**
6. **Employees whose salary is greater than FORD’s salary**

**1) SELECT ename**

**FROM emp**

**WHERE deptno = (**

**SELECT deptno**

**FROM dept**

**WHERE dname = 'SALES'**

**);**

**2) SELECT ename, sal**

**FROM emp**

**WHERE sal > (**

**SELECT AVG(sal)**

**FROM emp**

**);**

**3)SELECT ename**

**FROM emp**

**WHERE deptno = (**

**SELECT deptno**

**FROM emp**

**WHERE ename = 'SMITH'**

**);**

**4) SELECT ename, sal**

**FROM emp**

**WHERE sal = (**

**SELECT MAX(sal)**

**FROM emp**

**);**

**5) SELECT ename**

**FROM emp**

**WHERE deptno = (**

**SELECT deptno**

**FROM dept**

**WHERE loc = 'NEW YORK'**

**);**

**6) SELECT ename, sal**

**FROM emp**

**WHERE sal > (**

**SELECT sal**

**FROM emp**

**WHERE ename = 'FORD'**

**);**

**1.Students who enrolled in Python**

**2.Students who enrolled in any course**

**3.Students who did NOT enroll anywhere**

**4.Courses that cost more than average fee**

**5.Students mentored by student id = 1**

**6.Show which course have highest course fee**

**7.Show which course have lowest course fee**

**8.Students living in same city as “Ramesh”**

**9.Courses cheaper than Python course**

**10.Show students who joined a course after 1 Feb 2024**

**SELECT student\_name**

**FROM Students**

**WHERE student\_id IN (**

**SELECT student\_id**

**FROM Enrollments**

**WHERE course\_id = 101**

**);**

**SELECT student\_name**

**FROM Students**

**WHERE student\_id IN (**

**SELECT student\_id**

**FROM Enrollments**

**);**

**3)**

**SELECT student\_name**

**FROM Students**

**WHERE student\_id NOT IN (**

**SELECT student\_id FROM Enrollments**

**);**

**4)**

**SELECT course\_name**

**FROM Courses**

**WHERE fee > (**

**SELECT AVG(fee)**

**FROM Courses**

**);**

**5)**

**SELECT student\_name**

**FROM Students**

**WHERE mentor\_id = (**

**SELECT student\_id**

**FROM Students**

**WHERE student\_id = 1**

**);**

**6)**

**SELECT course\_name, fee**

**FROM Courses**

**WHERE fee = (**

**SELECT MAX(fee)**

**FROM Courses**

**);**

**7)**

**SELECT course\_name, fee**

**FROM Courses**

**WHERE fee = (**

**SELECT MIN(fee)**

**FROM Courses**

**);**

**8**

**SELECT student\_name**

**FROM Students**

**WHERE city = (**

**SELECT city**

**FROM Students**

**WHERE student\_name = 'Ramesh'**

**);**

**9.**

**SELECT course\_name**

**FROM Courses**

**WHERE fee < (**

**SELECT fee**

**FROM Courses**

**WHERE course\_name = 'Python'**

**);**

**10.**

**SELECT student\_name**

**FROM Students**

**WHERE student\_id IN (**

**SELECT student\_id**

**FROM Enrollments**

**WHERE enroll\_date > '2024-02-01'**

**);**

**TCL (Transaction Control Language)**

**What is TCL?**

Transaction Control Language (TCL) in MySQL is used to manage transactions in a database. Transactions are sequences of operations performed as a single logical unit of work, which must be either fully completed or fully rolled back.

TCL ensures data integrity and consistency in case of system failures or other unexpected issues.

**TCL Commands in MySQL**

1. **COMMIT** – Saves all changes made in the current transaction.
2. **ROLLBACK** – Undoes all changes made in the current transaction.
3. **SAVEPOINT** – Creates a temporary save point within a transaction.
4. **RELEASE SAVEPOINT** – Deletes a previously created save point.
5. **SET AUTOCOMMIT** – Controls the automatic commit behavior.

**COMMIT Command**

* COMMIT saves all the changes made in the current transaction permanently in the database.
* After a COMMIT, the changes cannot be undone.

-- Disable auto-commit mode

**SET AUTOCOMMIT = 0;**

-- Start a transaction

**START TRANSACTION;**

-- Insert new records

INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (101, 'John Doe', 50000);

INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (102, 'Jane Smith', 60000);

-- Commit the transaction (save changes)

**COMMIT;**

* The START TRANSACTION; command begins a transaction.
* Two records are inserted into the employees table.
* The COMMIT; command saves the changes permanently.

**ROLLBACK Command**

The ROLLBACK command undoes all the changes made in the current transaction before committing.

-- Start a new transaction

START TRANSACTION;

-- Insert new records

INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (103, 'Alice Brown', 55000);

INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (104, 'Bob White', 58000);

-- Rollback the transaction (undo changes)

**ROLLBACK;**

The records inserted within the transaction are **not saved** in the database because we executed ROLLBACK; instead of COMMIT;

SELECT \* FROM employees WHERE emp\_id = 103 OR emp\_id = 104;

🔹 **Expected Output:**  
*No records found*, as the rollback operation discarded the changes

**SAVEPOINT Command**

SAVEPOINT allows you to set a checkpoint in a transaction so that you can roll back only to that point instead of the entire transaction.

-- Start transaction

START TRANSACTION;

-- Insert first record

INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (105, 'David Miller', 62000);

-- Create a savepoint

**SAVEPOINT sp1;**

-- Insert second record

INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (106, 'Emma Watson', 65000);

-- Rollback to the savepoint (undo second insert but keep the first insert)

**ROLLBACK TO sp1;**

-- Commit the remaining transaction

COMMIT;

 The first insert (emp\_id = 105) is executed.

 A savepoint named sp1 is created.

 The second insert (emp\_id = 106) is executed.

 The rollback operation reverts changes made after SAVEPOINT sp1, undoing the second insert.

 The final commit saves only the first insert.

SELECT \* FROM employees WHERE emp\_id IN (105, 106);



Record for emp\_id = 106 does not exist because it was rolled back.

**RELEASE SAVEPOINT Command**

Deletes a savepoint but does not affect the transaction. Once released, a savepoint cannot be used for rollback.

**START TRANSACTION;**

**-- Insert a record**

**INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (107, 'Olivia Wilson', 67000);**

**-- Create savepoints**

**SAVEPOINT sp2;**

**INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (108, 'Daniel Brown', 70000);**

**-- Release savepoint**

**RELEASE SAVEPOINT sp2;**

**-- Try to rollback to sp2 (Will cause an error since it was released)**

**ROLLBACK TO sp2; -- Error: No such savepoint**

**COMMIT;**

* sp2 is created.
* After inserting another record, sp2 is released.
* Trying to roll back to sp2 results in an **error** because it no longer exists.

**SET AUTOCOMMIT Command**

* By default, MySQL automatically commits every operation.
* SET AUTOCOMMIT = 0; disables this behavior.
* SET AUTOCOMMIT = 1; enables auto-commit again.

**-- Disable auto-commit**

**SET AUTOCOMMIT = 0;**

**-- Start transaction**

**START TRANSACTION;**

**-- Insert a record**

**INSERT INTO employees (emp\_id, emp\_name, salary) VALUES (109, 'Sophia Carter', 72000);**

**-- Verify if the record is inserted**

**SELECT \* FROM employees WHERE emp\_id = 109; -- Record exists, but not committed**

**-- Enable auto-commit**

**SET AUTOCOMMIT = 1;**

 Before setting SET AUTOCOMMIT = 1;, the record is not permanently saved until a COMMIT; is executed.

 If MySQL restarts or a rollback happens before the commit, the inserted record is lost.

Data Control Language (DCL)

**1. Introduction to DCL**

Data Control Language (DCL) in MySQL is used to manage access rights and permissions for users. It consists of commands such as GRANT and REVOKE to provide or remove privileges from users.

**2. Creating Users**

To create a new user in MySQL, use the CREATE USER command:

**CREATE USER 'username'@'hostname' IDENTIFIED BY 'password';**

* username: The name of the user.
* hostname: Specifies the host from which the user can connect.
* password: The user's password.

**Examples:**

CREATE USER 'john'@'localhost' IDENTIFIED BY '123456';

CREATE USER 'jessy'@'localhost' IDENTIFIED BY 'dear123';

**To view all users:**

USE mysql;

SELECT user FROM user;

**To drop a user:**

DROP USER 'jessy'@'localhost';

**To check the current user:**

SELECT USER();

**3. Connecting as a Different User**

To switch users in MySQL:

\c username@localhost:3306

Example:

\c john@localhost:3306

**4. User Actions Verification**

1. **Show all databases accessible by a user:**

SHOW DATABASES;

1. **Show privileges granted to a user:**

SHOW PRIVILEGES;

1. **Show grants for a user:**

SHOW GRANTS FOR 'john'@'localhost';

**5. Granting Privileges**

To grant privileges to a user, use the GRANT command:

**GRANT privileges ON database\_name.table\_name TO 'username'@'hostname';**

**Examples:**

1. Grant specific privileges:

GRANT SELECT ON practice.emp TO 'john'@'localhost';

1. Grant all privileges on a specific database:

GRANT ALL ON practice.\* TO 'john'@'localhost';

1. Grant all privileges on all databases:

GRANT ALL ON \*.\* TO 'john'@'localhost';

1. To check granted privileges:

SHOW GRANTS FOR 'john'@'localhost';

**6. Revoking Privileges**

To revoke specific privileges from a user:

REVOKE privileges ON database\_name.table\_name FROM 'username'@'hostname';

**Example:**

REVOKE UPDATE ON practice.emp FROM 'jessy'@'localhost';

To verify the revoked privileges:

SHOW GRANTS FOR 'jessy'@'localhost';

**7. Changing Passwords**

To change a user's password:

SET PASSWORD FOR 'john'@'localhost' = 'newpassword';

Alternatively, using ALTER USER:

ALTER USER 'john'@'localhost' IDENTIFIED BY 'Preethi';

**8. Dropping a User**

To remove a user from MySQL:

DROP USER 'john'@'localhost';

Stored Procedures

**1. Introduction to Stored Procedures**

A **Stored Procedure** is a set of precompiled SQL statements stored in a database. It allows users to execute a sequence of SQL queries with a single command, improving efficiency and security.

**2. Advantages of Stored Procedures**

✅ **Performance Improvement:** Reduces query parsing and execution time.  
✅ **Security:** Prevents SQL injection and allows controlled access.  
✅ **Code Reusability:** Can be reused in multiple applications.  
✅ **Modularity:** Reduces redundancy by grouping SQL logic in one place.  
✅ **Maintainability:** Easier to manage and update complex SQL logic.

A **Stored Procedure** is a set of SQL statements stored in the database that can be executed repeatedly. It is primarily used for:

* **Reusability:** Acts like a function, allowing repeated execution.
* **Encapsulation:** Complex logic can be grouped together.
* **Single Unit of Work:** Can execute multiple SQL queries as a single operation.

**Features of Stored Procedures**

1. **Encapsulation:** Groups multiple SQL statements into a single procedure.
2. **Performance:** Reduces query parsing and improves execution speed.
3. **Security:** Can restrict user access to the underlying data.
4. **Maintainability:** Centralized SQL logic makes updates easier.

3. Creating a Stored Procedure

Syntax

**DELIMITER $$**

**CREATE PROCEDURE procedure\_name()**

**BEGIN**

**-- SQL Statements**

**END $$**

**DELIMITER ;**

Ex:-

DELIMITER $$

CREATE PROCEDURE getAllEmployees()

BEGIN

SELECT \* FROM emp;

END $$

DELIMITER ;

**Calling the Procedure**

**CALL getAllEmployees();**

**Dropping a Procedure**

**DROP PROCEDURE IF EXISTS getAllEmployees;**

**Checking Stored Procedures**

**SHOW PROCEDURE STATUS WHERE Db = "practice";**

**4. Using Input Parameters -** **Dynamic Stored Procedures**

Syntax

**CREATE PROCEDURE procedure\_name(IN parameterName DataType)**

**BEGIN**

**-- SQL Statement using the parameter**

**END;**

Ex:-

DELIMITER $$

CREATE PROCEDURE getEmpByID(IN emp\_id INT)

BEGIN

SELECT \* FROM emp WHERE empno = emp\_id;

END $$

DELIMITER ;

Calling with an Input Parameter

CALL getEmployeeByID(7839);

**5. Using Output Parameters in Stored Procedures**

Procedures can also **return values** using output parameters.

Syntax:-

**CREATE PROCEDURE procedure\_name(IN input\_param DataType, OUT output\_param DataType)**

**BEGIN**

**-- SQL Statement storing result into output\_param**

**END;**

Ex:- Getting Maximum Salary in a Department

DELIMITER $$

CREATE PROCEDURE getMaxSalary(IN dept\_no INT, OUT max\_sal DECIMAL(7,2))

BEGIN

SELECT MAX(sal) INTO max\_sal FROM emp WHERE deptno = dept\_no;

END $$

DELIMITER ;

Calling with an Output Parameter

CALL getMaxSalary(10, @output);

SELECT @output;

**6. Using Multiple Output Parameters**

We can return multiple values using multiple OUT parameters.

Ex:-

DELIMITER $$

CREATE PROCEDURE getMinMaxSalary(IN dept\_no INT, OUT max\_sal DECIMAL(7,2), OUT min\_sal DECIMAL(7,2))

BEGIN

SELECT MAX(sal) INTO max\_sal FROM emp WHERE deptno = dept\_no;

SELECT MIN(sal) INTO min\_sal FROM emp WHERE deptno = dept\_no;

END $$

DELIMITER ;

Calling with Multiple Outputs

CALL getMinMaxSalary(10, @max\_out, @min\_out);

SELECT @max\_out, @min\_out;

Views

**1. Introduction to Views**

A **view** is a **virtual table** that represents the result of a SQL query. It does **not store data** but dynamically retrieves it from underlying tables when queried.

**Key Features of Views:**

✅ **Virtual Table:** Acts like a table but doesn’t store data physically.  
✅ **Simplifies Complex Queries:** Helps in writing cleaner queries.  
✅ **Enhances Security:** Restricts access to specific columns or rows.  
✅ **Improves Code Readability:** Provides an abstraction layer over complex joins.

**2. Creating a View**

**Syntax:**

**CREATE VIEW view\_name AS**

**SELECT column1, column2, ...**

**FROM table\_name**

**WHERE condition;**

**Example 1: Creating a View for Employees in Department 10**

CREATE VIEW dept10\_employees AS

SELECT \* FROM emp WHERE deptno = 10;

✅ Now, the view dept10\_employees will contain employees only from department **10**.

**3. Querying a View**

Views can be queried just like normal tables.

**Example 2: Selecting Data from a View**

SELECT \* FROM dept10\_employees;

**Example 3: Performing Aggregations on a View**

SELECT MAX(sal) FROM dept10\_employees;

✅ This retrieves the highest salary from department **10**.

**4. Updating Data Through Views**

**⚠ Warning:** **Updating a view affects the underlying table!**

**Example 4: Updating a View**

UPDATE dept10\_employees SET sal = 1500 WHERE empno = 7369;

✅ This also updates the emp table.

**Restrictions on Updating Views:**

❌ Cannot update views that contain **aggregations (SUM, AVG, MAX, MIN, etc.)**  
❌ Cannot update views with **JOINS, GROUP BY, or DISTINCT**

**5. Modifying an Existing View**

If a view needs changes, we can modify it using **CREATE OR REPLACE VIEW**.

**Syntax:**

**CREATE OR REPLACE VIEW view\_name AS**

**SELECT column1, column2 FROM table\_name WHERE condition;**

**Example 5: Modifying a View**

CREATE OR REPLACE VIEW dept10\_employees AS

SELECT ename, sal FROM emp WHERE deptno = 10;

✅ Now, the view will show **only ename and sal**, instead of all columns.

**6. Dropping a View**

To remove a view, use the DROP VIEW statement.

**Syntax:**

**DROP VIEW view\_name;**

**Example 6: Dropping a View**

DROP VIEW dept10\_employees;

✅ The view is deleted, but the **original table remains** unaffected

**7. Advantages of Using Views**

✅ **Encapsulation:** Hides complex joins and queries behind simple names.  
✅ **Security:** Restricts access to certain columns or rows.  
✅ **Data Integrity:** Ensures users see a consistent dataset.  
✅ **Simplifies Queries:** No need to rewrite complex queries multiple times.

**8. Limitations of Views**

❌ **Cannot store data** (Only a virtual representation).  
❌ **Performance Overhead** (Each query is executed dynamically).  
❌ **Limited DML (INSERT, UPDATE, DELETE) support** (Not possible in certain cases).

**9. When to Use Views?**

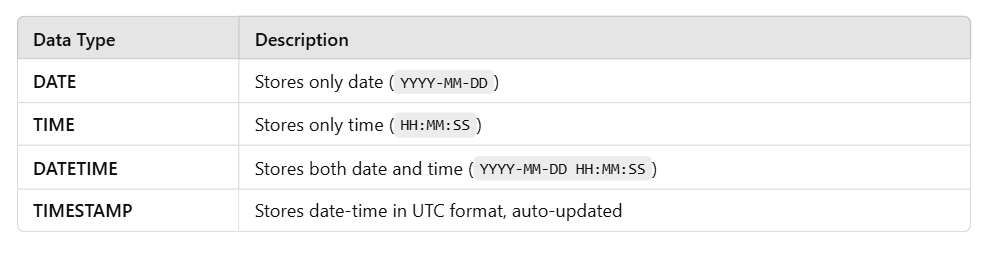
✔ When you want to **simplify complex queries**.  
✔ When you need **to restrict access** to specific columns or rows.  
✔ When creating a **reporting or analytics dashboard**.  
✔ When maintaining **backward compatibility** for database schema changes.

Date and Time Functions

**1. Introduction to Date and Time in MySQL**

MySQL provides various **date and time** functions to handle, manipulate, and format dates efficiently.

**Commonly Used Date and Time Data Types**



**2. Creating a Table with Date & Time Fields**

CREATE TABLE mytb (

mydate DATE,

mytime TIME,

mydatetime DATETIME

);

**Inserting Current Date and Time**

INSERT INTO mytb VALUES (CURRENT\_DATE(), CURRENT\_TIME(), NOW());

✅ **Shortcut Version:**

INSERT INTO mytb VALUES (CURDATE(), CURTIME(), NOW());

**3. Getting Tomorrow’s and Yesterday’s Date**

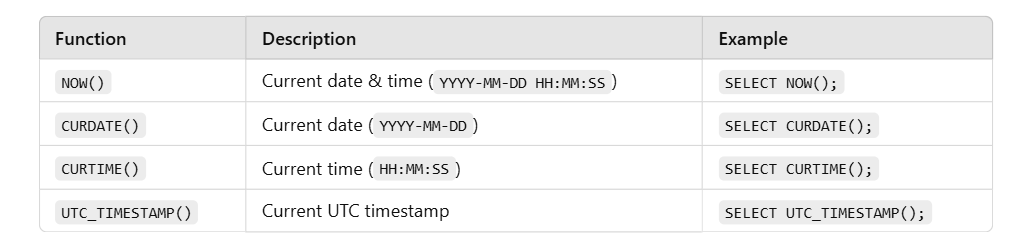
Insert Tomorrow's Date

INSERT INTO mytb VALUES (CURDATE() + INTERVAL 1 DAY, CURTIME(), NOW());

Insert Yesterday's Date

INSERT INTO mytb VALUES (CURDATE() - INTERVAL 1 DAY, CURTIME(), NOW());

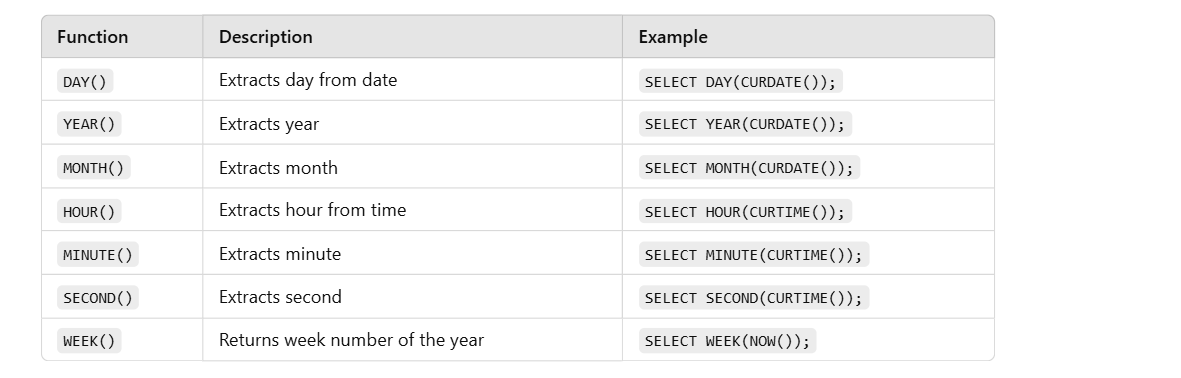
**4. Retrieving Current Date & Time**



Example Usage:

SELECT UTC\_TIMESTAMP(), NOW(), CURDATE(), CURTIME();

**5. Extracting Date and Time Components**



Example Query

SELECT DAY(CURDATE()), YEAR(CURDATE()), MONTH(CURDATE()),

HOUR(CURTIME()), MINUTE(CURTIME()), SECOND(CURTIME()), WEEK(NOW());

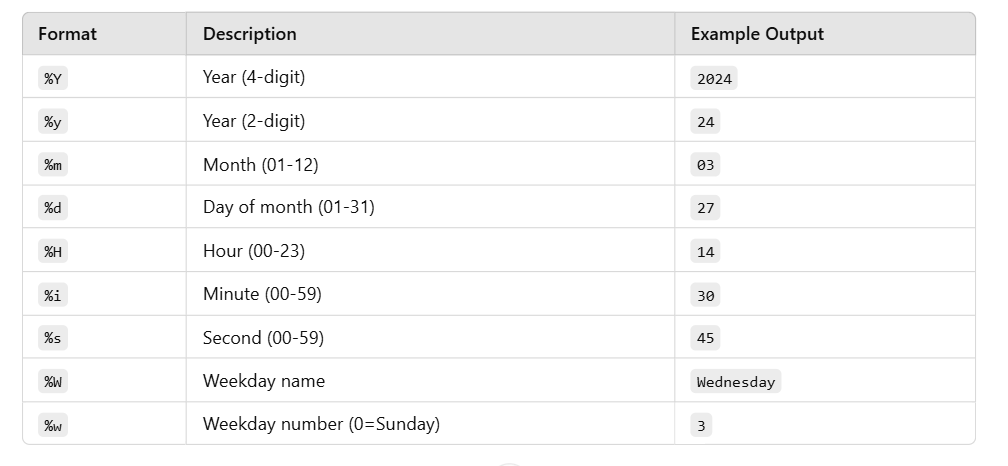
**6. Formatting Dates**

We can format date-time using DATE\_FORMAT().

Syntax:

SELECT DATE\_FORMAT(NOW(), 'format\_specifier');

Common Format Specifiers



Example 1: Custom Date Format

SELECT DATE\_FORMAT(NOW(), "%w-%d-%m-%y");

**7. Manipulating Dates (Date Arithmetic)**

Using DATE\_ADD() and DATE\_SUB() for adding/subtracting dates.

**Adding Days to a Date**

SELECT DATE\_ADD(CURDATE(), INTERVAL 20 DAY);

**Subtracting Days from a Date**

SELECT DATE\_SUB(CURDATE(), INTERVAL 5 DAY);

**8. Calculating Date Differences**

**Syntax:**

SELECT DATEDIFF(end\_date, start\_date);

Example 1: Days Between Two Dates

SELECT DATEDIFF('2024-11-24', '2024-10-15');

✅ Output:40

**9. Calculating Time Differences**

**Syntax:**

**SELECT TIMEDIFF(time1, time2);**

Example 2: Difference Between Two Time Values

SELECT TIMEDIFF(CURTIME(), '12:00:00');

✅ Output: Shows the time difference between the current time and 12:00:00.

**10. Summary of Date & Time Functions**

