**INTRODUCTION TO MYSQL**

**What is a Database?**

A database is an organized collection of data that can be easily accessed, managed, and updated. It stores data in a structured format, often using tables, to facilitate efficient retrieval and manipulation.

**Types of Databases**

* **Relational Database (SQL-based)** – MySQL, PostgreSQL, SQL Server (uses tables with relations).
* **NoSQL Database** – MongoDB, Firebase (uses documents, key-value pairs, etc.).

**What is a Relational Database?**

**A Relational Database (RDBMS - Relational Database Management** System) is a type of database that stores data in structured tables with rows and columns. The tables are related to each other through primary keys and foreign keys to maintain data integrity.

**Advantages of Relational Databases**

* Structured & Organized: Data is stored in a well-defined manner.
* Data Integrity: Enforces relationships using Primary Keys and Foreign Keys.
* Scalability: Can handle large datasets efficiently.
* Security: Supports user permissions and access control.

**INSTALLATION AND SETUP**

**Step 1: Download MySQL Installer**

* Go to the official MySQL website: [MySQL Installer](https://dev.mysql.com/downloads/installer/)
* Download Latest Version From this website.
* Choose the version for your operating system:
  + Windows (.msi)
  + macOS (.dmg)
  + Linux (.rpm/.deb)
* Click Download, then scroll down and select "No thanks, just start my download."
* Choosing a Setup as “custom”
* Go to MySQL Server 8.0 Drop Down -> Select Latest Version.
* Go to Applications Drop Down -> Select MySQL Workbench-> Select Latest Version.
* Go to MySQL Shell Drop Down -> Select MySQL Workbench-> Select Latest Version.
* Click “Execute” to install.

**Please refer below video for more clarity on the step involved in installation of MYSQL.**

<https://www.youtube.com/watch?v=uj4OYk5nKCg> **UNDERSTANDING DATABASES, TABLES AND RELATIONSHIPS**

**1. What is a Database?**

A database is an organized collection of data that stores information efficiently for easy access and management.

**Example:** A school database stores student records, courses, and teachers' details.

**2. What is a Table?**

A table is a structured way to store data inside a database using rows and columns.

**Students Table:**

| StudentID | Name | Age | Class |
| --- | --- | --- | --- |
| 1 | John | 15 | 10A |
| 2 | Alice | 14 | 9B |

**3. What is Relationship?**

Relationship defines how tables connect to each other using **Primary Keys (PK)** and **Foreign Keys (FK).**

**Enrolments Table:**

| **Enrolment\_ID** | **Student\_ID** | **Course\_ID** |
| --- | --- | --- |
| 1 | 1 | 101 |
| 2 | 2 | 102 |

**DATA TYPES IN MYSQL**

MySQL provides different data types to store various kinds of data efficiently. These data types are categorized into Numeric, String (Character), and Date & Time types.

1. **Numeric Data Types**

* Used to store numbers, including integers and decimals.
* Example - INT, FLOAT, DECIMAL

1. **String (Character) Data Types**

* Used to store text, characters, or binary data.
* Example - CHAR(n), VARCHAR(n), TEXT

1. **Date & Time Data Types**

* Used to store date, time, and timestamps.
* Example – DATE, DATETIME, TIMESTAMP, TIME & YEAR

**Choosing the Right Data Type**

✔ Use VARCHAR instead of CHAR for variable-length text.   
✔ Use TIMESTAMP for automatic date tracking.  
✔ Avoid TEXT for frequently queried data.

**MANIPULATING TABLES IN MYSQL**

1. **Creating a Table**

To create a table, use the CREATE TABLE statement.

**Example**: Creating a students table

**CREATE TABLE Students (**

**StudentID INT,**

**Name VARCHAR (50),**

**Age INT,**

**Class VARCHAR (10)**

**);**

1. **Inserting Data into a Table**

Use the INSERT INTO statement to add records.

**Example:** Adding a student detail into Students Table

**INSERT INTO Students (StudentID, Name, Age, Class)**

**VALUES**

**(1, 'Alice', 20, 'A', 'New York'),**

**(2, 'Bob', 22, 'B', 'Los Angeles'),**

**(3, 'Charlie', 21, 'A', 'Chicago'),**

**(4, 'David', 23, 'C', 'Houston'),**

**(5, 'Eve', 20, 'B', 'Phoenix');**

1. **Retrieving Data from a Table**

Use the SELECT statement to retrieve records.

**Example:** Get students details from students table

**SELECT \* FROM Students; // To get all the records**

**SELECT Name, Age FROM Students; // To get specific columns**

1. **Updating Data in Table**

Use the UPDATE statement to modify existing records.

**Example:** Updating a students age

**UPDATE Students SET Age = 16 WHERE StudentID = 1;**

1. **Deleting Data from Table**

Use the DELETE statement to remove records.

**Example:** Delete a record from students table

**DELETE FROM Students WHERE StudentID = 1;**

1. **Altering the Table**

Modify a table structure using ALTER TABLE.

**Example:** Add a new column

**ALTER TABLE Students ADD COLUMN Email VARCHAR (100);**

**Example:** Change a column data type

**ALTER TABLE Students MODIFY COLUMN Age TINYINT;**

**Example:** Delete a column

**ALTER TABLE Students DROP COLUMN Email;**

1. **Dropping a Table**

Use DROP TABLE to permanently delete a table.

**Example:** Delete the Students table

**DROP TABLE Students;**

1. **Truncating a Table**

Use TRUNCATE TABLE to delete all records while keeping the table structure.

**Example:** Truncate the data in Students table

**TRUNCATE TABLE Students;**

**Insert Data from another Table:**

Instead of manually adding values, you can insert data from another table.

**INSERT INTO Graduated\_Students (StudentID, Name, Age)**

**SELECT StudentID, Name, Age FROM Students WHERE Age > 18;**

**What is the SQL LIKE Operator?**

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column

% - Represents zero, one, or multiple characters.

\_ - Represents a single character.

SELECT \* FROM employees WHERE name LIKE '%\_son';

**SUMMARY**

✔ CREATE TABLE – Define table structure.  
✔ INSERT – Add new records.  
✔ SELECT – Retrieve data.  
✔ UPDATE – Modify existing records.  
✔ DELETE – Remove specific records.  
✔ ALTER TABLE – Modify table structure.  
✔ DROP TABLE – Permanently delete a table.  
✔ TRUNCATE TABLE – Delete all data but keep the table.

**CONSTRAINTS AND KEYS IN MYSQL**

Constraints in MySQL are rules applied to table columns to enforce data integrity and consistency. Keys are special constraints that define relationships between tables.

1. **PRIMARY KEY**

A Primary Key (PK) uniquely identifies each record in a table.

* Must be unique for each row.
* Cannot be NULL.

**CREATE TABLE Students (**

**StudentID INT PRIMARY KEY,**

**Name VARCHAR (50),**

**Age INT**

**);**

1. **FOREIGN KEY**

A Foreign Key (FK) establishes a relationship between two tables.

* References the Primary Key of another table.
* Ensures referential integrity.

**CREATE TABLE Enrolments (**

**EnrollmentID INT PRIMARY KEY,**

**StudentID INT,**

**CourseID INT,**

**FOREIGN KEY (StudentID) REFERENCES Students (StudentID),**

**FOREIGN KEY (CourseID) REFERENCES Courses (CourseID));**

1. **UNIQUE CONSTRAINT**

Ensures that all values in a column are distinct.

**CREATE TABLE Users (**

**UserID INT PRIMARY KEY,**

**Email VARCHAR (100) UNIQUE,**

**Username VARCHAR (50) UNIQUE**

**);**

1. **NOT NULL CONSTRAINT**

Ensures a column cannot store NULL values.

**CREATE TABLE Employees (**

**EmployeeID INT PRIMARY KEY,**

**Name VARCHAR (50) NOT NULL,**

**Salary DECIMAL (10,2) NOT NULL**

**);**

1. **CHECK CONSTRAINT**

Ensures that values in a column meet specific conditions.

**CREATE TABLE Products (**

**ProductID INT PRIMARY KEY,**

**Price DECIMAL (10,2) CHECK (Price > 0)**

**);**

1. **DEFAULT CONSTRAINT**

Sets a default value for a column when no value is provided.

**CREATE TABLE Orders (**

**OrderID INT PRIMARY KEY,**

**OrderDate DATE DEFAULT CURRENT\_DATE);**

1. **AUTO\_INCREMENT**

Automatically generates unique numbers for a Primary Key column.

**CREATE TABLE Customers (**

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

**Name VARCHAR (50)**

**);**

**SUMMARY OF CONSTRAINTS –**

| Constraint | Purpose |
| --- | --- |
| PRIMARY KEY | Uniquely identifies each row |
| FOREIGN KEY | Establishes relationships between tables |
| UNIQUE | Ensures unique values in a column |
| NOT NULL | Prevents NULL values in a column |
| CHECK | Enforces a condition on column values |
| DEFAULT | Assigns a default value if none is provided |
| AUTO\_INCREMENT | Automatically generates unique IDs |

**BASIC SQL COMMANDS FOR BEGINNERS -**

1. **CREATING A DATABASE**

**CREATE DATABASE my\_database;**

1. **USING A DATABASE**

**USE my\_database;**

1. **CREATING A TABLE**

**CREATE TABLE Students (**

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

**Name VARCHAR (50),**

**Age INT,**

**Class VARCHAR (10)**

**);**

1. **INSERTING DATA INTO A TABLE**

**INSERT INTO Students (Name, Age, Class) VALUES**

**('John Doe', 15, '10A'),**

**('Alice Smith', 14, '9B');**

1. **RETRIEVING DATA (SELECT)**

**SELECT \* FROM Students;**

1. **UPDATING DATA**

**UPDATE Students**

**SET Age = 16**

**WHERE Name = 'John Doe';**

1. **DELETING DATA**

**DELETE FROM Students WHERE Name = 'Alice Smith';**

1. **SORTING DATA (ORDER BY)**

**SELECT \* FROM Students ORDER BY Age DESC;**

1. **GROUPING DATA (GROUP BY & HAVING)**

**SELECT Class, COUNT (\*) AS Total\_Students**

**FROM Students**

**GROUP BY Class HAVING Total\_Students > 5;**

1. **FILTERING DATA (LIKE, BETWEEN, IN, LIMIT)**

**SELECT \* FROM Students WHERE Name LIKE 'J%';**

**SELECT \* FROM Students WHERE Age BETWEEN 14 AND 16;**

**SELECT \* FROM Students WHERE Class IN ('10A', '9B');**

**SELECT \* FROM Courses LIMIT 2;**

**SELECT \* FROM Courses LIMIT 1 offset 2;**

1. **CREATING RELATIONSHIPS (FOREIGN KEY)**

**CREATE TABLE Courses (**

**Course\_ID INT AUTO\_INCREMENT PRIMARY KEY,**

**Course\_Name VARCHAR (50));**

**CREATE TABLE Enrollments (**

**Enrollment\_ID INT AUTO\_INCREMENT PRIMARY KEY,**

**Student\_ID INT,**

**Course\_ID INT,**

**FOREIGN KEY (Student\_ID) REFERENCES Students (Student\_ID),**

**FOREIGN KEY (Course\_ID) REFERENCES Courses (Course\_ID)**

**);**

1. **DELETING A RECORD**

**DELETE TABLE TABLE\_NAME WHERE ID = 1;**

1. **DROP A DATABASE**

**DROP DATABASE my\_database;**

1. **DROP TABLE**

**DROP TABLE TABLE\_NAME;**

**AGGREGATE FUNCTIONS IN MYSQL**

Aggregate functions are used to perform calculations on multiple rows of a table and return a single value.

1. **COUNT ()**

Counts the number of rows in a column.

**SELECT COUNT (\*) FROM Students;**

**SELECT COUNT (DISTINCT StudentID) FROM Students;**

1. **SUM ()**

Returns the sum of the values in a specified column**.**

**SELECT SUM(Salary) FROM Employees;**

1. **AVG ()**

Returns the average value of a column.

**SELECT AVG(Age) FROM Students;**

1. **MIN ()**

Returns the minimum value in a column.

**SELECT MIN(Age) FROM Students;**

1. **MAX ()**

Returns the maximum value in a column**.**

**SELECT MAX(Salary) FROM Employees;**

1. **GROUP\_CONCAT ()**

Concatenates values from multiple rows into a single string. This is useful for combining values in a column into a list.

**SELECT GROUP\_CONCAT(Name) FROM Students;**

**JOINS IN MYSQL**

Joins in MySQL are used to combine data from two or more tables based on a related column. They help retrieve meaningful data that exists across multiple tables.

**Example Tables:** Consider below tables to understand join.

**1. Students Table**

|  |  |  |
| --- | --- | --- |
| **Student\_ID** | **NAME** | **AGE** |
| 1 | John Doe | 15 |
| 2 | Alice | 14 |
| 3 | Bob | 16 |
| 4 | Charlie | 17 |

**2. Enrolments Table**

|  |  |  |
| --- | --- | --- |
| **Enrollment\_ID** | **Student\_ID** | **Course\_ID** |
| 101 | 1 | C1 |
| 102 | 2 | C2 |
| 103 | 3 | C3 |
| 104 | 3 | C4 |
| 105 | 5 | C5 |

**3. Courses Table**

|  |  |
| --- | --- |
| **Course\_ID** | **Course\_Name** |
| C1 | Maths |
| C2 | Science |
| C3 | History |
| C4 | English |

**TYPES OF JOINS IN MYSQL:**

1. **Inner Join:**

Retrieves only matching records from both tables.

**Example:**

**SELECT Students.Name, Enrollments.Course\_ID**

**FROM Students**

**INNER JOIN Enrolments**

**ON Students.Student\_ID = Enrollments.Student\_ID;**

1. **Left Join**:

Returns all students, even if they haven't enrolled in a course (NULL if no match).

**Example:**

**SELECT Students.Name, Enrollments.Course\_ID**

**FROM Students**

**LEFT JOIN Enrolments**

**ON Students.Student\_ID = Enrollments.Student\_ID;**

1. **Right Join:**

Returns all courses, even if no students have enrolled in them (NULL if no match).

**Example:**

**SELECT Students.Name, Enrollments.Course\_ID**

**FROM Students**

**RIGHT JOIN Enrolments**

**ON Students.Student\_ID = Enrollments.Student\_ID;**

1. **Outer Join:**

Returns all students and all courses (NULL if no match).

**Example:**

**SELECT Students.Name, Courses.Course\_Name**

**FROM Students**

**LEFT JOIN Enrolments ON Students.Student\_ID = Enrollments.Student\_ID**

**UNION**

**SELECT Students.Name, Courses.Course\_Name**

**FROM Students**

**RIGHT JOIN Enrolments ON Students.Student\_ID = Enrollments.Student\_ID**

1. **Cross Join:**

Returns all possible student-course combinations.

**Example**:

**SELECT Students.Name, Courses.Course\_Name**

**FROM Students**

**CROSS JOIN Courses;**

1. **Self Join:**

A table is joined **with itself** to compare rows within the same table.

**Example:**

**SELECT A.Name AS Employee1, B.Name AS Employee2, A.Department**

**FROM Employees A**

**JOIN Employees B**

**ON A.Department = B.Department**

**AND A.Employee\_ID <> B.Employee\_ID;**

**SUBQUERIES AND NESTED QUERIES**

**What are Subqueries?**

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

A subquery is enclosed in parentheses () and is executed first before the main query.

**Types of Subqueries in MySQL**

**1. Single-Row Subquery**

* Returns only one value (single row & single column).
* Used with operators like =, <, >, etc.

SELECT Name, Age FROM Students WHERE Age > (SELECT MIN(Age) FROM Students);

**2. Multi-Row Subquery**

* + Returns multiple values (multiple rows but one column).
  + Used with IN, ANY, ALL operators.

SELECT Name FROM Students WHERE Student\_ID IN (SELECT Student\_ID FROM Enrolments WHERE Course\_ID = (SELECT Course\_ID FROM Enrolments WHERE Student\_ID = (SELECT Student\_ID FROM Students WHERE Name = 'Alice')));

1. **Correlated Subquery**

* Depends on the main query and runs once per row of the outer query.
* Used when filtering based on related data from another table.

SELECT Name FROM Students S WHERE (SELECT COUNT (\*) FROM Enrolments E WHERE E. Student\_ID = S. Student\_ID) > 1;

**INDEXING & PERFORMANCE OPTIMIZATION**

**What is Indexing?**

Indexes are critical for query performance, but they come with trade-offs. Let’s break down indexing strategies and best practices for MySQL.

**Syntax**: CREATE INDEX idx\_name ON Employees(Name);

**Types of Indexes in MySQL**

1. **Unique Index**

* Ensures that values are unique in a column.
* Faster than non-unique indexes because of uniqueness constraints.

**Example**: CREATE UNIQUE INDEX idx\_unique\_email ON Employees(Email);

1. **Composite** **Index (Multi-Column Index)**

* Index across multiple columns for faster queries.
* **Order matters** in composite indexes.

**Example**: CREATE INDEX idx\_name\_dept ON Employees(Name, Dept\_ID);

1. **Regular/Non-Unique Index**

* Indexing for particular column even with duplicate value.

**Example**: CREATE INDEX index\_name ON employees(name);

1. **Full-Text Index**

* Used for **fast text searches** (like searching in articles).

**Example**: CREATE FULLTEXT INDEX idx\_desc ON Products(Description);

**Checking and Analysing Indexes:**

1) To check Indexes on a Table:

**Syntax:** SHOW INDEX FROM Employees;

2) To analyse Query Performance:

**Syntax:** EXPLAIN SELECT \* FROM Employees WHERE Name = 'John';

**3**) Add Index to existing table:

**Syntax:** ALTER TABLE employees ADD INDEX index\_name(column\_name);

**When NOT to Use Indexes**

**❌ Avoid Indexes If:**

1. The table is small (scanning is faster than indexing).
2. The column has low cardinality (few unique values, e.g., Gender).
3. You frequently insert, update, or delete (indexes slow down writes).

**✅ Use Indexes for:**

1. Large tables
2. Queries with WHERE conditions
3. JOIN operations
4. Sorting & grouping

**Detailed Comparison:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Without Index** | **With Index** |
| Query Execution | Full Table Scan | Index Lookup and Row Fetch |
| Time Complexity | O(n) | O(log n) + O(k) |
| Disk I/O | High (reads the entire table) | Low (fetches only matching rows) |
| CPU Usage | High (evaluates condition for all rows) | Low (evaluates fewer rows) |
| Performance on Large Data | Poor | Excellent |
| Applicable for Small Tables | Doesn't matter (full scan is fast) | Overhead of maintaining an index may not justify its use |

**VIEWS AND STORED PROCEDURES**

**VIEWS IN MYSQL**

A view is a virtual table that stores a SQL query result. It simplifies complex queries and enhances security by restricting direct access to the underlying tables.

**Syntax:**

CREATE VIEW <view\_name> AS SELECT <column\_name> FROM <table>;

ALTER VIEW <view\_name AS SELECT <column\_name> FROM <table>;

**1) Creating a View:**

CREATE VIEW Employee\_View AS SELECT Emp\_ID, Name, Salary FROM Employees;

* Now, you can query it like a table:

SELECT \* FROM Employee\_View WHERE Salary > 50000;

**2) Updating a View:**

ALTER VIEW Employee\_View AS SELECT Emp\_ID, Name, Salary, Dept\_ID FROM Employees;

**3) Dropping a View:**

DROP VIEW Employee\_View;

**Key Benefits of Views:**

* Simplifies complex queries
* Enhances security by restricting table access
* Provides consistent reporting data

**STORED PROCEDURES:**

A stored procedure is a reusable set of SQL statements stored in the database, improving performance and security**.**

**Creating a Simple Stored Procedure:**

DELIMITER //

CREATE PROCEDURE GetEmployeeDetails(IN empID INT)

BEGIN

SELECT \* FROM Employees WHERE Emp\_ID = empID;

END //

DELIMITER ;

**Calling the Stored Procedure:**

CALL GetEmployeeDetails(102);

**Stored Procedure with Output Parameter:**

DELIMITER //

CREATE PROCEDURE GetEmployeeSalary(IN empID INT, OUT empSalary DECIMAL(10,2))

BEGIN

SELECT Salary INTO empSalary FROM Employees WHERE Emp\_ID = empID;

END //

DELIMITER ;

**Stored Procedure with IF Condition:**

DELIMITER //

CREATE PROCEDURE CheckEmployee(IN empID INT)

BEGIN

IF (SELECT COUNT(\*) FROM Employees WHERE Emp\_ID = empID) > 0 THEN

SELECT \* FROM Employees WHERE Emp\_ID = empID;

ELSE

SELECT 'No employee found' AS Message;

END IF;

END //

DELIMITER ;

**Dropping a Stored Procedure:**

DROP PROCEDURE GetEmployeeDetails;

**TRIGGERS AND EVENTS**

**TRIGGERS:**

A trigger is a special stored procedure that executes automatically before or after an event (INSERT, UPDATE, or DELETE) on a table.

**Trigger Syntax:**

CREATE TRIGGER trigger\_name

{BEFORE|AFTER} {INSERT|UPDATE|DELETE}

ON table\_name

FOR EACH ROW

trigger\_body;

**Explanation:**

* **trigger\_name**: The name of the trigger.
* **BEFORE|AFTER**: Whether the trigger will run before or after the event (insert, update, delete).
* **INSERT|UPDATE|DELETE**: Specifies which operation will trigger the action.
* **table\_name**: The table to which the trigger is attached.
* **FOR EACH ROW**: Indicates that the trigger will be executed once for each row affected by the operation.
* **trigger\_body**: The actual SQL statements that will be executed when the trigger is fired.

**Types of Triggers:**

1. **BEFORE Trigger:** Executes before the operation (INSERT, UPDATE, DELETE) occurs on the table.
2. **AFTER Trigger**: Executes after the operation is completed on the table.

In MySQL, when you're creating a trigger (or any stored procedure or function) that includes a block of code with multiple statements, you need to change the delimiter temporarily. This is because MySQL by default uses a semicolon (;) as a statement delimiter, and since your trigger or procedure will have multiple statements, it can cause a conflict.

To solve this, you use DELIMITER to temporarily change the statement delimiter while defining the trigger. Here's an example to demonstrate how to use DELIMITER with triggers:

**Example of Trigger with Delimiters:**

**AFTER**:

-- Change delimiter to something else, like //

DELIMITER //

CREATE TRIGGER log\_new\_customer

AFTER INSERT

ON customers

FOR EACH ROW

BEGIN

INSERT INTO audit\_log (action, table\_name, record\_id, created\_at)

VALUES ('INSERT', 'customers', NEW.customer\_id, NOW());

END//

-- Reset delimiter back to semicolon

DELIMITER ;

**BEFORE**:

-- Change delimiter to something else, like //

DELIMITER //

CREATE TRIGGER before\_customer\_insert

BEFORE INSERT

ON customers

FOR EACH ROW

BEGIN

-- Example logic: Set a default value if customer\_name is NULL

IF NEW.customer\_name IS NULL THEN

SET NEW.customer\_name = 'Default Name';

END IF;

END//

-- Reset delimiter back to semicolon

DELIMITER ;

**EVENTS:**

An event is a scheduled task that runs automatically at a specified time interval. It is similar to a cron job in Linux.

✅ Enabling the Event Scheduler

SET GLOBAL event\_scheduler = ON;

✅ Dropping an Event

DROP EVENT Apply\_Bonus;

✅ Creating a Recurring Event

DELIMITER //

CREATE EVENT Delete\_Inactive\_Users

ON SCHEDULE EVERY 1 DAY

DO

BEGIN

DELETE FROM Users WHERE Last\_Login < DATE\_SUB(NOW(), INTERVAL 6 MONTH);

END //

DELIMITER ;

✅ Creating a One-Time Event

DELIMITER //

CREATE EVENT Apply\_Bonus

ON SCHEDULE AT '2025-01-01 00:00:00'

DO

BEGIN

UPDATE Employees SET Salary = Salary \* 1.10;

END //

DELIMITER ;

|  |  |  |
| --- | --- | --- |
| **Feature** | **Triggers** | **Events** |
| **Execution** | Fires **on table changes** (INSERT, UPDATE, DELETE) | Runs **on a scheduled basis** |
| **Use Case** | Audit logs, data validation | Automated cleanup, scheduled updates |
| **Example** | Log salary changes | Delete inactive users daily |

**TRANSACTIONS AND ERROR HANDLING**

**TRANSACTIONS:**

A transaction is a sequence of one or more SQL statements that execute as a single unit.

**Commit Example:**

START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE Account\_ID = 1;

UPDATE Accounts SET Balance = Balance + 500 WHERE Account\_ID = 2;

COMMIT;

**Rolling Back a Transaction:**

START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE Account\_ID = 1;

SELECT 1 / 0;

UPDATE Accounts SET Balance = Balance + 500 WHERE Account\_ID = 2;

ROLLBACK;

**Using SAVEPOINTS:**

START TRANSACTION;

UPDATE Employees SET Salary = Salary + 1000 WHERE Emp\_ID = 1;

SAVEPOINT BeforeBonus;

UPDATE Employees SET Bonus = Bonus + 500 WHERE Emp\_ID = 1;

ROLLBACK TO BeforeBonus;

COMMIT;

**ERROR HANDLING**

Error handling in MySQL is done using DECLARE HANDLER, which catches specific errors and prevents transaction failures.

**✅ Handling Errors with EXIT HANDLER**

DELIMITER //

CREATE PROCEDURE TransferMoney(IN fromAcc INT, IN toAcc INT, IN amount DECIMAL(10,2))

BEGIN

DECLARE EXIT HANDLER FOR SQLEXCEPTION

BEGIN

ROLLBACK;

SELECT 'Transaction Failed!' AS Message;

END;

START TRANSACTION;

UPDATE Accounts SET Balance = Balance - amount WHERE Account\_ID = fromAcc;

UPDATE Accounts SET Balance = Balance + amount WHERE Account\_ID = toAcc;

COMMIT;

SELECT 'Transaction Successful!' AS Message;

END //

DELIMITER ;

**✅ Handling Errors with CONTINUE HANDLER**

CONTINUE allows execution to continue after handling an error.

DELIMITER //

CREATE PROCEDURE UpdateSalary(IN empID INT, IN newSalary DECIMAL(10,2))

BEGIN

DECLARE CONTINUE HANDLER FOR SQLEXCEPTION

SET @error = 'Salary Update Failed';

UPDATE Employees SET Salary = newSalary WHERE Emp\_ID = empID;

SELECT IFNULL(@error, 'Salary Updated Successfully') AS Status;

END //

DELIMITER ;

**USER MANAGEMENT AND SECURITY**

✅ **Creating a New User**

CREATE USER 'john\_doe'@'localhost' IDENTIFIED BY 'SecurePass123';

✅ **Granting Permissions**

GRANT SELECT, INSERT, UPDATE ON company\_db.\* TO 'john\_doe'@'localhost';

✅ **Granting All Privileges (Use with caution!)**

GRANT ALL PRIVILEGES ON company\_db.\* TO 'admin\_user'@'localhost';

✅ **Revoking Permissions**

REVOKE INSERT, UPDATE ON company\_db.\* FROM 'john\_doe'@'localhost';

✅ **Deleting a User**

DROP USER 'john\_doe'@'localhost';

**JSON DATA IN MYSQL**

**CREATING A TABLE WITH JSON DATA**

CREATE TABLE Employees (Emp\_ID INT PRIMARY KEY, Name VARCHAR(100), Details JSON);

**Inserting JSON Data**

INSERT INTO Employees (Emp\_ID, Name, Details) VALUES(1, 'Alice', '{"Department": "HR", "Skills": ["Communication", "Recruitment"]}');

**Retrieving JSON Data**

* SELECT \* FROM Employees;
* SELECT Details->>'$.Department' AS Department FROM Employees;
* SELECT JSON\_EXTRACT(Details, '$.Skills[0]') AS First\_Skill FROM Employees;

**Search in JSON Data**

* SELECT \* FROM jsonexample WHERE JSON\_CONTAINS(Details, '"HR"', '$.Department');
* SELECT \* FROM jsonexample WHERE JSON\_SEARCH(Details, 'one', 'Recruitment') IS NOT NULL;

**Update in JSON Data**

* UPDATE jsonexample SET Details = JSON\_SET(Details, '$.Experience', 5) WHERE Emp\_ID = 3;
* UPDATE jsonexample SET Details = JSON\_REPLACE(Details, '$.Department', 'Finance')
* WHERE Emp\_ID = 3;
* UPDATE jsonexample SET Details = JSON\_REMOVE(Details, '$.Skills') WHERE Emp\_ID = 5;

**PARTITIONING TABLES**

Table partitioning in MySQL allows you to divide a large table into smaller, more manageable pieces, called partitions. These partitions allow for improved performance and manageability of large datasets.

**Types of Partitioning**

1. Range Partitioning
2. List Partitioning
3. **RANGE PARTITIONING:**

**Range Partitioning** divides a table into partitions based on a **range of values**. This is useful when you want to separate data by some ordered value (e.g., dates or numeric ranges).

CREATE TABLE Sales (

Sale\_ID INT,

Sale\_Date DATE,

Amount DECIMAL(10,2)

)

PARTITION BY RANGE (YEAR(Sale\_Date)) (

PARTITION p\_2019 VALUES LESS THAN (2020),

PARTITION p\_2020 VALUES LESS THAN (2021),

PARTITION p\_2021 VALUES LESS THAN (2022)

);

1. **LIST PARTITIONING:**

List Partitioning divides the table into partitions based on a specific list of values.

CREATE TABLE Employees (

Emp\_ID INT,

Name VARCHAR(100),

Region VARCHAR(50)

)

PARTITION BY LIST columns (Region) (

PARTITION p\_north VALUES IN ('North'),

PARTITION p\_south VALUES IN ('South'),

PARTITION p\_east VALUES IN ('East'),

PARTITION p\_west VALUES IN ('West')

);

**Insert Data into the Partitioned Table:**

When you insert data into this partitioned table, MySQL will automatically place each row in the correct partition based on the partition.

INSERT INTO Sales (Sale\_ID, Sale\_Date, Amount)

VALUES (1, '2020-05-15', 150.00),

(2, '2020-05-16', 150.00),

(3, '2021-05-17', 150.00);

**To Check Partition in particular table:**

select \* from INFORMATION\_SCHEMA.PARTITIONS where table\_name = "Sales";

**Full-Text Search**

Full-Text Search (FTS) in MySQL is used for performing advanced search queries on text-based data stored in CHAR, VARCHAR, or TEXT columns.

**Creating a Table with a Full-Text Index:**

CREATE TABLE articles (

id INT AUTO\_INCREMENT PRIMARY KEY,

title VARCHAR(255),

content TEXT,

FULLTEXT (title, content) -- Creating a Full-Text Index

);

**Inserting Sample Data:**

INSERT INTO articles (title, content) VALUES

('MySQL Full-Text Search', 'This article explains how full-text search works in MySQL.'),

('SQL Indexing Techniques', 'Indexing is important for database optimization and performance.'),

('Introduction to MySQL', 'MySQL is a powerful relational database management system.');

**Performing Full-Text Searches:**

**Basic Full-Text Search**

SELECT \* FROM articles

WHERE MATCH(title, content) AGAINST('MySQL');

**Searching for Multiple Words**

SELECT \* FROM articles

WHERE MATCH(title, content) AGAINST('MySQL Search');

**Normalization Techniques in Databases**

Normalization in databases is the process of organizing data to reduce redundancy and improve data integrity. It involves dividing a database into multiple related tables and applying rules to ensure data is stored efficiently. There are several normalization forms (NF), each addressing specific issues:

**1. First Normal Form (1NF)**

* Ensures atomicity (each column contains only indivisible values).
* Removes duplicate columns from the same table.
* Identifies each row uniquely using a primary key.

**Example (Before 1NF)**

| **StudentID** | **Name** | **Courses** |
| --- | --- | --- |
| **1** | **John** | **Math, Science** |
| **2** | **Alice** | **English, History** |

**After 1NF (Separate rows for each course)**

| **StudentID** | **Name** | **Course** |
| --- | --- | --- |
| **1** | **John** | **Math** |
| **1** | **John** | **Science** |
| **2** | **Alice** | **English** |
| **2** | **Alice** | **History** |

**2. Second Normal Form (2NF)**

* Must be in 1NF.
* Removes partial dependencies (i.e., a non-key attribute should depend on the whole primary key, not just part of it).
* Typically applies to composite primary keys.

**Example (Before 2NF)**

| **StudentID** | **CourseID** | **StudentName** | **CourseName** |
| --- | --- | --- | --- |
| **1** | **101** | **John** | **Math** |
| **1** | **102** | **John** | **Science** |

**After 2NF (Separate into two tables)**

**Students Table**

| **StudentID** | **StudentName** |
| --- | --- |
| **1** | **John** |
| **2** | **Alice** |

**Courses Table**

| **CourseID** | **CourseName** |
| --- | --- |
| **101** | **Math** |
| **102** | **Science** |

**Student\_Courses Table (Mapping Table)**

| **StudentID** | **CourseID** |
| --- | --- |
| **1** | **101** |
| **1** | **102** |
| **2** | **103** |

**3. Third Normal Form (3NF)**

* Must be in 2NF.
* Removes transitive dependencies (i.e., non-key attributes should depend only on the primary key, not other non-key attributes).

**Example (Before 3NF)**

| **StudentID** | **StudentName** | **CourseID** | **CourseName** | **Instructor** | **InstructorOffice** |
| --- | --- | --- | --- | --- | --- |
| **1** | **John** | **101** | **Math** | **Dr. Smith** | **Room 201** |
| **1** | **John** | **102** | **Science** | **Dr. Lee** | **Room 202** |

**After 3NF (Separate into different tables)**

**Courses Table**

| **CourseID** | **CourseName** | **Instructor** |
| --- | --- | --- |
| **101** | **Math** | **Dr. Smith** |
| **102** | **Science** | **Dr. Lee** |

**Instructors Table**

| **Instructor** | **InstructorOffice** |
| --- | --- |
| **Dr. Smith** | **Room 201** |
| **Dr. Lee** | **Room 202** |

**4. Boyce-Codd Normal Form (BCNF)**

* Must be in 3NF.
* Ensures every determinant is a candidate key (i.e., no non-trivial functional dependency exists on anything other than a superkey).

**Example (Before BCNF)**

| **StudentID** | **CourseID** | **Instructor** |
| --- | --- | --- |
| **1** | **101** | **Dr. Smith** |
| **2** | **101** | **Dr. Smith** |
| **3** | **102** | **Dr. Lee** |

**After BCNF (Separate Instructor Assignment)**

**Course\_Instructor Table**

| **CourseID** | **Instructor** |
| --- | --- |
| **101** | **Dr. Smith** |
| **102** | **Dr. Lee** |

**Student\_Courses Table remains the same.**

**5. Fourth Normal Form (4NF)**

* Must be in BCNF.
* Eliminates multi-valued dependencies (i.e., when one attribute determines multiple independent attributes).

**Example (Before 4NF)**

| **EmployeeID** | **Skill** | **Project** |
| --- | --- | --- |
| **1** | **Java** | **A** |
| **1** | **Python** | **A** |
| **1** | **Java** | **B** |
| **1** | **Python** | **B** |

**After 4NF (Separate into two tables)**

**Employee\_Skills Table**

| **EmployeeID** | **Skill** |
| --- | --- |
| **1** | **Java** |
| **1** | **Python** |

**Employee\_Projects Table**

| **EmployeeID** | **Project** |
| --- | --- |
| **1** | **A** |
| **1** | **B** |

**6. Fifth Normal Form (5NF)**

* Must be in 4NF.
* Deals with join dependencies, ensuring no unnecessary joins are needed to reconstruct data.

**7. Sixth Normal Form (6NF)**

* Deals with temporal databases and non-trivial join dependencies.
* Rarely used in practical applications.

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

**Each normalization step improves data integrity and efficiency but may require more tables and joins. In practice, databases are usually normalized up to 3NF or BCNF, as further normalization may not always be necessary.**