## Exercise 6: View Creation and Manipulation

**Aim:**
To create and manipulate database views in MySQL for simplifying complex queries and improving data security.

**Procedure:**

1. Create a sample database and an `Employee` table.
2. Insert sample employee data into the table.
3. Create different views to display specific information.
4. Modify an existing view.
5. Drop the created view.

## Step 1: Create Database and Table

```
CREATE DATABASE CompanyDB;
USE CompanyDB;

CREATE TABLE Employee (
    EmpID INT AUTO_INCREMENT PRIMARY KEY,
    EmpName VARCHAR(50),
    Department VARCHAR(50),
    Salary DECIMAL(10,2),
    Location VARCHAR(50)
);
```

## Step 2: Insert Sample Data

```
INSERT INTO Employee (EmpName, Department, Salary, Location) VALUES
('Arun', 'IT', 50000, 'Chennai'),
('Priya', 'HR', 45000, 'Coimbatore'),
('Kiran', 'Finance', 60000, 'Madurai'),
('Meena', 'IT', 55000, 'Chennai'),
('Deepa', 'Sales', 48000, 'Trichy');
```

---

## Step 3: Create a Simple View

```
CREATE VIEW IT_Employees AS
SELECT EmpID, EmpName, Salary
FROM Employee
WHERE Department = 'IT';
```

**Query 1 Output (View Data):**

```
SELECT * FROM IT_Employees;
```

**Expected Output:**

| EmpID | EmpName | Salary |
|---|---|---|
| 1 | Arun | 50000.00 |
| 4 | Meena | 55000.00 |

---

## Step 4: Create a View with Calculated Columns

```
CREATE VIEW Employee_SalaryInfo AS
SELECT EmpName, Department, Salary,
       (Salary * 0.10) AS Bonus,
       (Salary + (Salary * 0.10)) AS TotalPay
FROM Employee;
```

**Query 2 Output:**

```
SELECT * FROM Employee_SalaryInfo;
```

**Expected Output:**

| EmpName | Department | Salary | Bonus | TotalPay |
|---|---|---|---|---|
| Arun | IT | 50000.00 | 5000.00 | 55000.00 |
| Priya | HR | 45000.00 | 4500.00 | 49500.00 |
| Kiran | Finance | 60000.00 | 6000.00 | 66000.00 |
| Meena | IT | 55000.00 | 5500.00 | 60500.00 |
| Deepa | Sales | 48000.00 | 4800.00 | 52800.00 |

---

## Step 5: Update the Data Using View

```
UPDATE IT_Employees
SET Salary = Salary + 2000
WHERE EmpName = 'Arun';
```

**Query 3 Output:**

```
SELECT * FROM IT_Employees;
```

**Expected Output (After Update):**

| EmpID | EmpName | Salary |
|---|---|---|
| 1 | Arun | 52000.00 |
| 4 | Meena | 55000.00 |

---

## Step 6: Modify (Replace) the Existing View

```
CREATE OR REPLACE VIEW IT_Employees AS
SELECT EmpName, Salary, Location
FROM Employee
WHERE Department = 'IT';
```

### Query 4 Output:

```
SELECT * FROM IT_Employees;
```

### Expected Output:

| EmpName | Salary | Location |
|---|---|---|
| Arun Kumar | 52000.00 | Chennai |
| Meena Roy | 55000.00 | Chennai |

---

## Step 7: Drop a View

```
DROP VIEW Employee_SalaryInfo;
```

### Query 5 Output:

```
SHOW FULL TABLES IN CompanyDB WHERE TABLE_TYPE LIKE 'VIEW';
```

### Expected Output:

| View Name |
|---|
| IT_Employees |

---

## Result:

- Successfully created and managed database views.
- Learned how to simplify complex queries using views.
- Demonstrated updating and modifying views dynamically.
- Verified data security and abstraction through view operations.

# Exercise 7: Flow Control Management in MySQL

## Aim:

To demonstrate flow control constructs in MySQL such as **IF, CASE, and loops** for conditional data manipulation and reporting.

## Procedure:

1. Create a sample database and table for employees' performance data.
2. Insert sample records into the table.
3. Use **IF statements** to classify employees based on salary.
4. Use **CASE statements** to provide performance grading.
5. Optionally, use **loops (WHILE/REPEAT)** in stored procedures to process data.
6. Verify outputs after each query or procedure execution.

## Step 1: Create Database and Table

```
CREATE DATABASE FlowControlDB;
USE FlowControlDB;

CREATE TABLE EmployeePerformance (
    EmpID INT AUTO_INCREMENT PRIMARY KEY,
    EmpName VARCHAR(50),
    Department VARCHAR(50),
    Salary DECIMAL(10,2),
    PerformanceScore INT
);
```

## Step 2: Insert Sample Data

```
INSERT INTO EmployeePerformance (EmpName, Department, Salary,
PerformanceScore) VALUES
('Arun', 'IT', 50000, 85),
('Priya', 'HR', 45000, 70),
('Kiran', 'Finance', 60000, 95),
('Meena', 'IT', 55000, 60),
('Deepa', 'Sales', 48000, 75);
```

---

## Step 3: IF Statement Example

```
SELECT EmpName, Salary,
       IF(Salary > 50000, 'High Salary', 'Low/Medium Salary') AS SalaryLevel
FROM EmployeePerformance;
```

**Expected Output:**

| EmpName | Salary | SalaryLevel |
|---|---|---|
| Arun | 50000.00 | Low/Medium Salary |
| Priya | 45000.00 | Low/Medium Salary |
| Kiran | 60000.00 | High Salary |
| Meena | 55000.00 | High Salary |
| Deepa | 48000.00 | Low/Medium Salary |

---

## Step 4: CASE Statement Example

```
SELECT EmpName, PerformanceScore,
       CASE
           WHEN PerformanceScore >= 85 THEN 'Excellent'
           WHEN PerformanceScore >= 70 THEN 'Good'
           WHEN PerformanceScore >= 50 THEN 'Average'
           ELSE 'Poor'
       END AS PerformanceGrade
FROM EmployeePerformance;
```

**Expected Output:**

| EmpName | PerformanceScore | PerformanceGrade |
|---|---|---|
| Arun | 85 | Excellent |
| Priya | 70 | Good |
| Kiran | 95 | Excellent |
| Meena | 60 | Average |
| Deepa | 75 | Good |

---

## Step 5: WHILE Loop Example (Stored Procedure)

```
DELIMITER $$

CREATE PROCEDURE IncreaseSalary()
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE total INT;
    SELECT COUNT(*) INTO total FROM EmployeePerformance;

    WHILE i <= total DO
        UPDATE EmployeePerformance
        SET Salary = Salary + 2000
        WHERE EmpID = i;
        SET i = i + 1;
    END WHILE;
END$$

DELIMITER ;
```

```
-- Execute the procedure
CALL IncreaseSalary();
```

### Query to Verify Update:

```
SELECT EmpName, Salary FROM EmployeePerformance;
```

### Expected Output:

| EmpName | Salary |
|---|---|
| Arun | 52000.00 |
| Priya | 47000.00 |
| Kiran | 62000.00 |
| Meena | 57000.00 |
| Deepa | 50000.00 |

---

## Result:

- Successfully used **IF statements** to classify salary levels.
- Applied **CASE statements** to grade employee performance.
- Created a **stored procedure with a WHILE loop** to update salaries dynamically.
- Learned practical **flow control management** techniques in MySQL for data analysis and manipulation.

## Exercise 8: Cursors, Joins, Triggers, and Functions

### Aim:

To demonstrate the use of **cursors, joins, triggers, and user-defined functions** in MySQL for handling complex queries and automated operations.

### Procedure:

1. Create a sample database and tables for employees and departments.
2. Insert sample data into the tables.
3. Demonstrate **inner join and left join** to combine data.
4. Use a **cursor** to iterate through a result set.
5. Create a **trigger** to automatically update a log table after insertion.
6. Create a **user-defined function** to calculate bonuses.
7. Verify the results after each operation.

### Step 1: Create Database and Tables

```
CREATE DATABASE CompanyDB2;
USE CompanyDB2;

CREATE TABLE Department (
    DeptID INT AUTO_INCREMENT PRIMARY KEY,
    DeptName VARCHAR(50)
);

CREATE TABLE Employee (
    EmpID INT AUTO_INCREMENT PRIMARY KEY,
    EmpName VARCHAR(50),
    DeptID INT,
    Salary DECIMAL(10,2),
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
);

CREATE TABLE SalaryLog (
    LogID INT AUTO_INCREMENT PRIMARY KEY,
    EmpID INT,
    OldSalary DECIMAL(10,2),
    NewSalary DECIMAL(10,2),
    ChangeDate DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

---

### Step 2: Insert Sample Data

```
INSERT INTO Department (DeptName) VALUES
('IT'), ('HR'), ('Finance');

INSERT INTO Employee (EmpName, DeptID, Salary) VALUES
('Arun', 1, 50000),
('Priya', 2, 45000),
('Kiran', 3, 60000),
('Meena', 1, 55000),
('Deepa', 3, 48000);
```

---

## Step 3: Joins

**Inner Join:** Employees with their department names

```
SELECT e.EmpName, e.Salary, d.DeptName
FROM Employee e
INNER JOIN Department d ON e.DeptID = d.DeptID;
```

**Expected Output:**

| EmpName | Salary | DeptName |
|---|---|---|
| Arun | 50000 | IT |
| Priya | 45000 | HR |
| Kiran | 60000 | Finance |
| Meena | 55000 | IT |
| Deepa | 48000 | Finance |

**Left Join:** Show all departments even if no employees

```
SELECT d.DeptName, e.EmpName
FROM Department d
LEFT JOIN Employee e ON d.DeptID = e.DeptID;
```

**Expected Output:**

| DeptName | EmpName |
|---|---|
| IT | Arun |
| IT | Meena |
| HR | Priya |
| Finance | Kiran |
| Finance | Deepa |

---

## Step 4: Cursor Example

```
DELIMITER $$

CREATE PROCEDURE PrintEmployees()
BEGIN
    DECLARE done INT DEFAULT 0;
    DECLARE eName VARCHAR(50);
    DECLARE eSalary DECIMAL(10,2);
    DECLARE cur CURSOR FOR SELECT EmpName, Salary FROM Employee;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN cur;

    read_loop: LOOP
        FETCH cur INTO eName, eSalary;
        IF done THEN
            LEAVE read_loop;
        END IF;
        SELECT CONCAT('Employee: ', eName, ' | Salary: ', eSalary) AS
EmployeeInfo;
    END LOOP;

    CLOSE cur;
END$$

DELIMITER ;

-- Execute cursor procedure
CALL PrintEmployees();
```

**Expected Output:**

| EmployeeInfo |
|---|
| Employee: Arun |
| Employee: Priya |
| Employee: Kiran |
| Employee: Meena |
| Employee: Deepa |

---

## Step 5: Trigger Example

Automatically log salary changes

```
DELIMITER $$

CREATE TRIGGER SalaryUpdateTrigger
AFTER UPDATE ON Employee
FOR EACH ROW
BEGIN
    IF OLD.Salary <> NEW.Salary THEN
        INSERT INTO SalaryLog(EmpID, OldSalary, NewSalary)
        VALUES (NEW.EmpID, OLD.Salary, NEW.Salary);
    END IF;
END$$
```

```
DELIMITER ;

-- Update salary to test trigger
UPDATE Employee SET Salary = Salary + 2000 WHERE EmpID = 1;
SELECT * FROM SalaryLog;
```

**Expected Output (SalaryLog):**

| LogID | EmpID | OldSalary | NewSalary | ChangeDate |
|---|---|---|---|---|
| 1 | 1 | 50000 | 52000 | 2025-10-24 14:00:00 |

---

### Step 6: User-Defined Function Example

Calculate 10% bonus for employees

```
DELIMITER $$

CREATE FUNCTION CalcBonus(salary DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    RETURN salary * 0.10;
END$$

DELIMITER ;

-- Use the function
SELECT EmpName, Salary, CalcBonus(Salary) AS Bonus
FROM Employee;
```

**Expected Output:**

| EmpName | Salary | Bonus |
|---|---|---|
| Arun | 52000 | 5200 |
| Priya | 45000 | 4500 |
| Kiran | 60000 | 6000 |
| Meena | 55000 | 5500 |
| Deepa | 48000 | 4800 |

---

## Result:

- Successfully performed **inner join and left join** queries.
- Implemented a **cursor** to iterate through records.
- Created a **trigger** to automatically log salary changes.
- Defined a **user function** to calculate bonuses.
- Learned practical **flow control, automation, and modularity** in MySQL.

# Exercise 9: Stored Procedure in MySQL

## Aim:

To create and execute a **stored procedure** in MySQL for performing repetitive database operations such as inserting, updating, or retrieving data.

## Procedure:

1. Create a sample database and a table for employees.
2. Insert some sample employee data.
3. Create a stored procedure to perform a specific task (e.g., retrieve employees of a department or update salary).
4. Execute the stored procedure.
5. Verify the output of the procedure.

## Step 1: Create Database and Table

```
CREATE DATABASE StoredProcDB;
USE StoredProcDB;

CREATE TABLE Employee (
    EmpID INT AUTO_INCREMENT PRIMARY KEY,
    EmpName VARCHAR(50),
    Department VARCHAR(50),
    Salary DECIMAL(10,2)
);
```

## Step 2: Insert Sample Data

```
INSERT INTO Employee (EmpName, Department, Salary) VALUES
('Arun', 'IT', 50000),
('Priya', 'HR', 45000),
('Kiran', 'Finance', 60000),
('Meena', 'IT', 55000),
('Deepa', 'Sales', 48000);
```

## Step 3: Create a Stored Procedure

### Example 1: Retrieve all employees of a specific department

```
DELIMITER $$

CREATE PROCEDURE GetEmployeesByDept(IN deptName VARCHAR(50))
BEGIN
    SELECT EmpID, EmpName, Department, Salary
    FROM Employee
    WHERE Department = deptName;
END$$
```

```
DELIMITER ;
```

## Step 4: Execute the Stored Procedure

```
CALL GetEmployeesByDept('IT');
```

### Expected Output:

| EmpID | EmpName | Department | Salary |
|---|---|---|---|
| 1 | Arun | IT | 50000 |
| 4 | Meena | IT | 55000 |

## Step 5: Create Another Stored Procedure (Update Salary)

```
DELIMITER $$

CREATE PROCEDURE GiveRaise(IN empID INT, IN raiseAmount DECIMAL(10,2))
BEGIN
    UPDATE Employee
    SET Salary = Salary + raiseAmount
    WHERE EmpID = empID;
END$$

DELIMITER ;
```

## Step 6: Execute the Update Procedure

```
CALL GiveRaise(1, 2000);
SELECT * FROM Employee WHERE EmpID = 1;
```

### Expected Output:

| EmpID | EmpName | Department | Salary |
|---|---|---|---|
| 1 | Arun | IT | 52000 |

---

## Result:

- Successfully created stored procedures for **retrieving and updating data**.
- Learned to **pass input parameters** to stored procedures.
- Simplified repetitive tasks and improved **modularity and maintainability** in MySQL.

# Ex 10: Graph Database Modeling: Person Residence Pattern using Cypher Queries

Aim

To create a simple graph model in Neo4j representing people and their cities of residence using:

Nodes : Person and City
Relationship : LIVES_IN (directed from Person to City)
Properties : Names, ages, population, and residence duration

**Procedure**

Basic Steps Followed:

**Step 1:** Access Neo4j Environment

Opened Neo4j Browser (Desktop/Sandbox)
Connected to the database instance

**Step 2:** Database Preparation

Cleared existing data to start fresh

**Step 3:** Node Creation

Created Person nodes with properties (name, age)
Created City nodes with properties (name, population)

**Step 4:** Relationship Establishment

Created directed LIVES_IN relationships from Person to City
Added temporal property (since) to relationships

**Step 5:** Data Verification

Executed various queries to validate the graph structure
Visualized the graph relationships

Queries

**Query 1: Database Clearance**

```
MATCH (n) DETACH DELETE n;
```

Expected Output: Database cleared (no nodes/relationships)

**Query 2: Create All Nodes and Relationships (Single Command)**

```
CREATE (p1:Person {name: "Gopi", age: 30}),
       (p2:Person {name: "Kavi", age: 25}),
       (p3:Person {name: "Kohul", age: 35}),
       (c1:City {name: "Thanjavur", population: 222943}),
       (c2:City {name: "Trichy", population: 916857}),
       (c3:City {name: "Chennai", population: 7088000}),
       (p1)-[:LIVES_IN {since: 2018}]->(c1),
       (p2)-[:LIVES_IN {since: 2020}]->(c2),
       (p3)-[:LIVES_IN {since: 2015}]->(c3);
```

Expected Output:

Created 3 Person nodes, 3 City nodes, 3 LIVES_IN relationships

**Query 3. View Complete Graph**

```
MATCH (n) RETURN n;
```

Expected Output:
Visual graph showing 6 nodes and 3 relationships

**Query 4. Query All People with Their Cities**

```
MATCH (p:Person)-[l:LIVES_IN]->(c:City)
RETURN p.name AS Person, p.age AS Age, c.name AS City,
       c.population AS Population, l.since AS "Living
       Since";
```

Expected Output Table:

| Person | Age | City | Population | Living Since |
|---|---|---|---|---|
| Gopi | 30 | Thanjavur | 222,943 | 2018 |
| Kavi | 25 | Trichy | 916,857 | 2020 |
| Kohul | 35 | Chennai | 7,088,000 | 2015 |

**Query 5. Find People Living in Specific City**

```
MATCH (p:Person)-[:LIVES_IN]->(c:City {name: "Chennai"})
RETURN p.name AS Resident, p.age AS Age;
```

Expected Output:

| Resident | Age |
|---|---|
| Kohul | 35 |

**Query 6. Count Residents per City**

```
MATCH (p:Person)-[:LIVES_IN]->(c:City)
RETURN c.name AS City, COUNT(p) AS Residents
ORDER BY Residents DESC;
```

Expected Output:

| City | Residents |
|---|---|
| Thanjavur | 1 |
| Trichy | 1 |
| Chennai | 1 |

**Query 7. Find Longest Resident**

```
MATCH (p:Person)-[l:LIVES_IN]->(c:City)
RETURN p.name AS Person, c.name AS City, l.since AS "Since
Year"
ORDER BY l.since ASC
LIMIT 1;
```

Expected Output:

| Person | City | Since Year |
|---|---|---|
| Kohul | Chennai | 2015 |

**Result**

The exercise successfully demonstrated basic Neo4j concepts including node creation, relationship establishment, property assignment, and graph querying using Cypher language.