Contents

Database Normalization

Example: Database Normalization in MySQL, from unnormalized data to 3rd Normal Form (3NF).

Step 1: Unnormalized Table (UNF)

```
CREATE DATABASE IF NOT EXISTS normalization_demo;
USE normalization_demo;

DROP TABLE IF EXISTS orders_unnormalized;

CREATE TABLE orders_unnormalized (
  order_id INT,
  customer_name VARCHAR(100),
  customer_address VARCHAR(255),
  product_1_name VARCHAR(100),
  product_1_qty INT,
  product_2_name VARCHAR(100),
  product_2_qty INT
);

INSERT INTO orders_unnormalized VALUES
(1, 'John Smith', '123 Elm St', 'Laptop', 1, 'Mouse', 2),
(2, 'Alice Brown', '456 Oak St', 'Monitor', 1, 'Keyboard', 1);
```

Step 2: First Normal Form (1NF)

Remove repeating groups (e.g., multiple product columns). Use atomic values (1 piece of data per field).

```
DROP TABLE IF EXISTS orders_1nf;

CREATE TABLE orders_1nf (
  order_id INT,
  customer_name VARCHAR(100),
  customer_address VARCHAR(255),
  product_name VARCHAR(100),
  product_qty INT
);

INSERT INTO orders_1nf VALUES
(1, 'John Smith', '123 Elm St', 'Laptop', 1),
(1, 'John Smith', '123 Elm St', 'Mouse', 2),
(2, 'Alice Brown', '456 Oak St', 'Monitor', 1),
(2, 'Alice Brown', '456 Oak St', 'Keyboard', 1);
```

Step 3: Second Normal Form (2NF)
Eliminate partial dependencies (i.e., fields depending on part of a composite primary key).

Separate Customer and Product entities.

```
-- Customers Table
DROP TABLE IF EXISTS customers;
CREATE TABLE customers (
  customer_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_name VARCHAR(100),
  customer_address VARCHAR(255)
);

-- Products Table
DROP TABLE IF EXISTS products;
CREATE TABLE products (
  product_id INT PRIMARY KEY AUTO_INCREMENT,
  product_name VARCHAR(100)
);

-- Orders Table
DROP TABLE IF EXISTS orders;
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  customer_id INT,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

-- OrderDetails Table
DROP TABLE IF EXISTS order_details;
CREATE TABLE order_details (
  order_id INT,
  product_id INT,
  product_qty INT,
  PRIMARY KEY (order_id, product_id),
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id)
);

-- Insert data
INSERT INTO customers (customer_name, customer_address) VALUES
('John Smith', '123 Elm St'),
('Alice Brown', '456 Oak St');
```

```
INSERT INTO products (product_name) VALUES
('Laptop'), ('Mouse'), ('Monitor'), ('Keyboard');

INSERT INTO orders (order_id, customer_id) VALUES
(1, 1), (2, 2);

INSERT INTO order_details (order_id, product_id, product_qty) VALUES
(1, 1, 1),
(1, 2, 2),
(2, 3, 1),
(2, 4, 1);
```

Step 4: Third Normal Form (3NF)
Remove transitive dependencies (non-key fields depending on other non-key fields).
In this case, already achieved: every non-key depends on the key, the whole key, and nothing but the key.

Indexing and Constraints

```
-- Create demo database
CREATE DATABASE IF NOT EXISTS indexing_constraints_demo;
USE indexing_constraints_demo;

-- Drop tables if they already exist
DROP TABLE IF EXISTS employees;
DROP TABLE IF EXISTS departments;

-- Create departments table
CREATE TABLE departments (
  department_id INT PRIMARY KEY AUTO_INCREMENT,
  department_name VARCHAR(100) UNIQUE NOT NULL
);

-- Create employees table
CREATE TABLE employees (
  employee_id INT PRIMARY KEY AUTO_INCREMENT,              -- Primary Key
  employee_name VARCHAR(100) NOT NULL,                     -- NOT NULL
  email VARCHAR(100) UNIQUE,                               -- UNIQUE constraint
  salary DECIMAL(10,2) CHECK (salary >= 0),                -- CHECK constraint
  department_id INT,                                       -- FK column
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,          -- DEFAULT constraint
  FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

```
-- Add sample departments
INSERT INTO departments (department_name) VALUES
('HR'),
('Finance'),
('Engineering');

-- Add sample employees
INSERT INTO employees (employee_name, email, salary, department_id) VALUES
('Alice Smith', 'alice@example.com', 50000.00, 1),
('Bob Johnson', 'bob@example.com', 70000.00, 2),
('Charlie Brown', 'charlie@example.com', 80000.00, 3);
```

View Constraints in the Table Inspector (MySQL Workbench)

- Go to the Schemas tab, right-click your database → Refresh All.
- Expand the employees and departments tables.
- Right-click a table → Table Inspector:
- Go to the Indexes tab → View primary, unique, and foreign key indexes.
- Go to the Columns tab → Check NOT NULL, DEFAULT, and CHECK constraints.

Add Additional Indexes (Manual Demo)

Add a single-column index:

```
CREATE INDEX idx_employee_name ON employees(employee_name);
```

Add a multi-column (composite) index:

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

To confirm:
Refresh schema → Right-click employees → Table Inspector → Indexes tab.

Validation Queries (Optional)

Show all indexes on a table

```
SHOW INDEX FROM employees;
```

Show all constraints

```
SELECT
  TABLE_NAME,
  CONSTRAINT_NAME,
  CONSTRAINT_TYPE
FROM
  information_schema.TABLE_CONSTRAINTS
WHERE
  TABLE_SCHEMA = 'indexing_constraints_demo';
```

Table Relationships, Joins and Union

```
-- Create and use the demo database
CREATE DATABASE IF NOT EXISTS joins_unions_demo;
USE joins_unions_demo;

-- Drop tables if they exist
DROP TABLE IF EXISTS order_items;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS customers;
DROP TABLE IF EXISTS products;

-- Create Customers table
CREATE TABLE customers (
  customer_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_name VARCHAR(100),
  city VARCHAR(100)
);

-- Create Products table
CREATE TABLE products (
  product_id INT PRIMARY KEY AUTO_INCREMENT,
  product_name VARCHAR(100),
  price DECIMAL(10,2)
);

-- Create Orders table
CREATE TABLE orders (
  order_id INT PRIMARY KEY AUTO_INCREMENT,
  customer_id INT,
  order_date DATE,
  FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

-- Create Order Items table (many-to-many)
CREATE TABLE order_items (
  order_item_id INT PRIMARY KEY AUTO_INCREMENT,
  order_id INT,
  product_id INT,
  quantity INT,
  FOREIGN KEY (order_id) REFERENCES orders(order_id),
  FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

```
-- Insert sample customers
INSERT INTO customers (customer_name, city) VALUES
('Alice', 'New York'),
('Bob', 'Los Angeles'),
('Charlie', 'Chicago');

-- Insert sample products
INSERT INTO products (product_name, price) VALUES
('Laptop', 1200.00),
('Mouse', 25.50),
('Monitor', 300.00);

-- Insert sample orders
INSERT INTO orders (customer_id, order_date) VALUES
(1, '2024-01-10'),
(2, '2024-02-15'),
(3, '2024-03-05');

-- Insert sample order items
INSERT INTO order_items (order_id, product_id, quantity) VALUES
(1, 1, 1),
(1, 2, 2),
(2, 3, 1),
(3, 2, 1);
```

View Relationships

a. Right-click the schema > Create EER Diagram.
b. Add all 4 tables.
c. View auto-drawn relationships via foreign keys.

Demonstrate JOINs

INNER JOIN (only matching rows)

```
SELECT
  c.customer_name,
  o.order_id,
  p.product_name,
  oi.quantity
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id;
```

LEFT JOIN (all customers, with/without orders)

```
SELECT
  c.customer_name,
  o.order_id
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id;
```

RIGHT JOIN (all orders even if customer deleted — not shown here)

```
SELECT
  c.customer_name,
  o.order_id
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;
```

FULL OUTER JOIN (Simulated using UNION)
*mysql have no full outer join but can be achieved by using union with left + right joins

```
SELECT
  c.customer_name,
  o.order_id
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id

UNION

SELECT
  c.customer_name,
  o.order_id
FROM customers c
RIGHT JOIN orders o ON c.customer_id = o.customer_id;
```

CROSS JOIN (Cartesian Product)

```
SELECT
  c.customer_name,
  p.product_name
FROM customers c
CROSS JOIN products p;
```

Demonstrate UNION and UNION ALL
UNION: Combine two customer name lists without duplicates

```
SELECT customer_name FROM customers
UNION
SELECT customer_name FROM customers WHERE city = 'Chicago';
```

UNION ALL: Combine two lists including duplicates

```
SELECT customer_name FROM customers
UNION ALL
SELECT customer_name FROM customers WHERE city = 'Chicago';
```

Validation Queries

View foreign keys:

```
SELECT
  TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME, REFERENCED_TABLE_NAME
FROM
  information_schema.KEY_COLUMN_USAGE
WHERE
  TABLE_SCHEMA = 'joins_unions_demo' AND REFERENCED_TABLE_NAME IS NOT NULL;
```

SQL functions

```
-- Create demo database
CREATE DATABASE IF NOT EXISTS sql_functions_demo;
USE sql_functions_demo;

-- Drop table if it exists
DROP TABLE IF EXISTS employees;

-- Create employees table
CREATE TABLE employees (
  emp_id INT PRIMARY KEY AUTO_INCREMENT,
  full_name VARCHAR(100),
  department VARCHAR(50),
  salary DECIMAL(10, 2),
  hire_date DATE,
  bonus_percent DECIMAL(5,2)
);

-- Insert sample data
INSERT INTO employees (full_name, department, salary, hire_date, bonus_percent) VALUES
('Alice Smith', 'Finance', 65000.50, '2020-01-15', 5.00),
('Bob Johnson', 'Engineering', 80000.00, '2019-06-10', 10.00),
('Charlie Brown', 'HR', 45000.75, '2021-11-05', 3.00),
('David Lee', 'Finance', 72000.00, '2018-08-20', 4.50),
('Eva Green', 'Engineering', 95000.00, '2023-03-01', NULL),
('Frank Black', 'HR', 50000.00, '2022-05-30', 7.00);
```

Aggregate Functions

SUM(), AVG(), MAX(), MIN(), COUNT()

```
-- Total salary of all employees
SELECT SUM(salary) AS total_salary FROM employees;

-- Average salary per department
SELECT department, AVG(salary) AS avg_salary FROM employees GROUP BY department;

-- Highest and lowest salary
SELECT MAX(salary) AS max_salary, MIN(salary) AS min_salary FROM employees;

-- Count of employees per department
SELECT department, COUNT(*) AS emp_count FROM employees GROUP BY department;
```

Scalar Functions

UPPER(), LOWER(), LENGTH(), NOW(), IFNULL()

```
-- Convert names to uppercase
SELECT full_name, UPPER(full_name) AS upper_name FROM employees;

-- Show length of each name
SELECT full_name, LENGTH(full_name) AS name_length FROM employees;

-- Show current date with name
SELECT full_name, NOW() AS current_time FROM employees;

-- Handle NULL bonus_percent with IFNULL
SELECT full_name, IFNULL(bonus_percent, 0) AS bonus_percent FROM employees;
```

String Functions

CONCAT(), SUBSTRING(), INSTR(), REPLACE()

```
-- Concatenate name and department
SELECT CONCAT(full_name, ' works in ', department) AS employee_info FROM employees;

-- Extract first name
SELECT full_name, SUBSTRING(full_name, 1, INSTR(full_name, ' ') - 1) AS first_name FROM employees;

-- Replace 'HR' with 'Human Resources'
SELECT department, REPLACE(department, 'HR', 'Human Resources') AS new_dept FROM employees;
```

Numeric Functions

ROUND(), CEIL(), FLOOR(), MOD(), ABS()

```
-- Round salary
SELECT full_name, salary, ROUND(salary, 0) AS rounded_salary FROM employees;

-- Ceil and Floor salary
SELECT full_name, CEIL(salary) AS ceiling_salary, FLOOR(salary) AS floor_salary FROM employees;

-- Modulo on salary
SELECT full_name, MOD(salary, 1000) AS salary_mod_1000 FROM employees;

-- Absolute value of bonus (simulate negative bonus)
SELECT full_name, ABS(IFNULL(bonus_percent, -5)) AS abs_bonus FROM employees;
```

Date Functions

YEAR(), MONTH(), DATEDIFF(), CURDATE(), DATE_FORMAT()

```
-- Extract hire year
SELECT full_name, YEAR(hire_date) AS hire_year FROM employees;

-- Number of days since hired
SELECT full_name, DATEDIFF(CURDATE(), hire_date) AS days_with_company FROM employees;

-- Format hire date
SELECT full_name, DATE_FORMAT(hire_date, '%M %d, %Y') AS formatted_hire_date FROM employees;

-- Show who was hired in the current year
SELECT * FROM employees WHERE YEAR(hire_date) = YEAR(CURDATE());
```

More Tasks:

- Show all employees who earn more than the average salary.
- List employees and their total compensation (salary + salary * bonus_percent).
- Format a string: UPPER(first_name) + "_" + hire_year
- Who has been in the company for more than 3 years?

Custom Stored Functions

User-defined functions (UDFs) or stored functions

*Reuse employees from earlier

Calculate Annual Bonus

Purpose: Return bonus amount based on salary and bonus percent.

```
DELIMITER //

DROP FUNCTION IF EXISTS calculate_bonus;

CREATE FUNCTION calculate_bonus(salary DECIMAL(10,2), bonus_percent DECIMAL(5,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
  DECLARE bonus_amount DECIMAL(10,2);
  SET bonus_amount = salary * (IFNULL(bonus_percent, 0) / 100);
  RETURN bonus_amount;
END;
//

DELIMITER ;
```

Format Full Name as Uppercase

```
DELIMITER //

DROP FUNCTION IF EXISTS format_name;

CREATE FUNCTION format_name(name_input VARCHAR(100))
RETURNS VARCHAR(100)
DETERMINISTIC
BEGIN
  RETURN UPPER(name_input);
END;
//

DELIMITER ;
```

Get Employee Tenure in Years

```
DELIMITER //

DROP FUNCTION IF EXISTS get_tenure_years;

CREATE FUNCTION get_tenure_years(hire_date DATE)
RETURNS INT
DETERMINISTIC
```

```
BEGIN
  RETURN TIMESTAMPDIFF(YEAR, hire_date, CURDATE());
END;
//

DELIMITER ;
```

Use Custom Functions in SELECT Queries

```
-- 1. Calculate bonus
SELECT
  full_name,
  salary,
  bonus_percent,
  calculate_bonus(salary, bonus_percent) AS bonus_amount
FROM employees;
```

```
-- 2. Format names
SELECT
  full_name,
  format_name(full_name) AS upper_name
FROM employees;
```

```
-- 3. Get tenure
SELECT
  full_name,
  hire_date,
  get_tenure_years(hire_date) AS years_with_company
FROM employees;
```

Notes for Demonstration in MySQL Workbench

- Paste each CREATE FUNCTION block separately.
- Use the "DELIMITER" command to avoid errors with semicolons inside functions.
- Refresh schema → Look under "Functions" in the left navigation panel to see them.

Drop Custom Functions

```
DROP FUNCTION IF EXISTS calculate_bonus;
DROP FUNCTION IF EXISTS format_name;
DROP FUNCTION IF EXISTS get_tenure_years;
```

Subqueries, Operators and Derived tables

```
CREATE DATABASE IF NOT EXISTS subquery_demo;
```

```
USE subquery_demo;

-- Drop existing tables
DROP TABLE IF EXISTS employees;
DROP TABLE IF EXISTS departments;

-- Create departments table
CREATE TABLE departments (
  department_id INT PRIMARY KEY AUTO_INCREMENT,
  department_name VARCHAR(100)
);

-- Create employees table
CREATE TABLE employees (
  employee_id INT PRIMARY KEY AUTO_INCREMENT,
  full_name VARCHAR(100),
  department_id INT,
  salary DECIMAL(10,2),
  hire_date DATE,
  FOREIGN KEY (department_id) REFERENCES departments(department_id)
);

-- Insert departments
INSERT INTO departments (department_name) VALUES
('HR'), ('Finance'), ('Engineering');

-- Insert employees
INSERT INTO employees (full_name, department_id, salary, hire_date) VALUES
('Alice Smith', 1, 45000.00, '2020-01-10'),
('Bob Johnson', 2, 60000.00, '2019-05-20'),
('Charlie Brown', 2, 58000.00, '2022-03-15'),
('David Lee', 3, 80000.00, '2021-07-01'),
('Eva Green', 3, 95000.00, '2023-01-01');
```

Subqueries (Single Row)
Show employees who earn more than the average salary

```
SELECT full_name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

Subqueries (Multiple Rows) with IN

Show employees in departments that have more than 1 employee

```
SELECT full_name
FROM employees
WHERE department_id IN (
  SELECT department_id
  FROM employees
  GROUP BY department_id
  HAVING COUNT(*) > 1
);
```

Correlated Subquery

List employees whose salary is greater than the average salary of their department

```
SELECT full_name, salary, department_id
FROM employees e1
WHERE salary > (
  SELECT AVG(salary)
  FROM employees e2
  WHERE e1.department_id = e2.department_id
);
```

EXISTS vs. IN

Show departments that have at least one employee (using EXISTS)

```
SELECT department_name
FROM departments d
WHERE EXISTS (
  SELECT 1
  FROM employees e
  WHERE e.department_id = d.department_id
);
```

To show the same result using IN:

```
SELECT department_name
FROM departments
WHERE department_id IN (SELECT department_id FROM employees);
```

Use of ANY and ALL

Find employees who earn more than any HR employee

```
SELECT full_name, salary
FROM employees
WHERE salary > ANY (
  SELECT salary FROM employees WHERE department_id = 1
);
```

Find employees who earn more than all HR employees

```
SELECT full_name, salary
FROM employees
WHERE salary > ALL (
  SELECT salary FROM employees WHERE department_id = 1
);
```

Derived Table (Inline View)
Show department average salary using a derived table and filter those with avg salary > 60000

```
SELECT *
FROM (
  SELECT d.department_name, AVG(e.salary) AS avg_salary
  FROM employees e
  JOIN departments d ON e.department_id = d.department_id
  GROUP BY d.department_name
) AS dept_avg
WHERE avg_salary > 60000;
```

Subquery in SELECT Clause
Show each employee with department average salary

```
SELECT
  full_name,
  salary,
  department_id,
  (SELECT AVG(salary) FROM employees e2 WHERE e1.department_id = e2.department_id) AS
dept_avg_salary
FROM employees e1;
```

Window Functions

```
CREATE DATABASE IF NOT EXISTS window_demo;
USE window_demo;

-- Drop table if it exists
DROP TABLE IF EXISTS sales;

-- Create a sales table
CREATE TABLE sales (
  sale_id INT AUTO_INCREMENT PRIMARY KEY,
  salesperson VARCHAR(100),
  region VARCHAR(50),
  sale_date DATE,
  sale_amount DECIMAL(10,2)
```

```
);

-- Insert sample sales data
INSERT INTO sales (salesperson, region, sale_date, sale_amount) VALUES
('Alice', 'North', '2024-01-01', 1000),
('Alice', 'North', '2024-01-05', 1500),
('Bob', 'North', '2024-01-07', 1200),
('Bob', 'North', '2024-01-10', 1700),
('Charlie', 'South', '2024-01-01', 900),
('Charlie', 'South', '2024-01-02', 1600),
('David', 'South', '2024-01-05', 1100),
('David', 'South', '2024-01-09', 1300);
```

Aggregate Window Function
Cumulative total per salesperson

```
SELECT
  salesperson,
  sale_date,
  sale_amount,
  SUM(sale_amount) OVER (PARTITION BY salesperson ORDER BY sale_date) AS running_total
FROM sales;
```

Ranking Window Functions
Rank sales by amount per region

```
SELECT
  salesperson,
  region,
  sale_date,
  sale_amount,
  RANK() OVER (PARTITION BY region ORDER BY sale_amount DESC) AS region_rank,
  DENSE_RANK() OVER (PARTITION BY region ORDER BY sale_amount DESC) AS dense_rank,
  ROW_NUMBER() OVER (PARTITION BY region ORDER BY sale_amount DESC) AS row_num
FROM sales;
```

Navigation Functions: LAG and LEAD
Compare current sale with previous and next sale for each salesperson

```
SELECT
  salesperson,
  sale_date,
  sale_amount,
  LAG(sale_amount, 1) OVER (PARTITION BY salesperson ORDER BY sale_date) AS previous_sale,
  LEAD(sale_amount, 1) OVER (PARTITION BY salesperson ORDER BY sale_date) AS next_sale
FROM sales;
```

First and Last Value
First and last sale per region (based on date)

```
SELECT
  salesperson,
  region,
  sale_date,
  sale_amount,
  FIRST_VALUE(sale_amount) OVER (PARTITION BY region ORDER BY sale_date) AS first_sale,
  LAST_VALUE(sale_amount) OVER (PARTITION BY region ORDER BY sale_date ROWS BETWEEN
UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS last_sale
FROM sales;
```

More tasks:

- Highest single-day sale per salesperson using RANK().
- Difference in sales between current and previous entry using sale_amount - LAG(...).
- Find top 2 sales per region using RANK() in a subquery.

MySQL Workbench Tips

- Run each section independently.
- View results to understand how the OVER() clause partitions and orders rows.
- You can also create views from these queries to visualize repeated logic.

Stored Procedures

CREATE DATABASE IF NOT EXISTS procedure_demo;

```
USE procedure_demo;

-- Drop if exists
DROP TABLE IF EXISTS employees;

-- Create table
CREATE TABLE employees (
  emp_id INT AUTO_INCREMENT PRIMARY KEY,
  full_name VARCHAR(100),
  department VARCHAR(50),
  salary DECIMAL(10, 2),
  hire_date DATE
);

-- Insert sample data
INSERT INTO employees (full_name, department, salary, hire_date) VALUES
('Alice Smith', 'HR', 45000, '2020-01-10'),
('Bob Johnson', 'Finance', 60000, '2019-06-15'),
('Charlie Brown', 'Engineering', 72000, '2021-03-01');
```

Simple Stored Procedure (No Parameters)

```
DELIMITER //

DROP PROCEDURE IF EXISTS GetAllEmployees;

CREATE PROCEDURE GetAllEmployees()
BEGIN
  SELECT * FROM employees;
END;
//

DELIMITER ;
```

```
-- Call it:
CALL GetAllEmployees();
```

Stored Procedure with IN Parameter

```
DELIMITER //

DROP PROCEDURE IF EXISTS GetEmployeesByDept;

CREATE PROCEDURE GetEmployeesByDept(IN dept_name VARCHAR(50))
BEGIN
  SELECT * FROM employees WHERE department = dept_name;
END;
//

DELIMITER ;
```

```
-- Call it:
CALL GetEmployeesByDept('HR');
```

Stored Procedure with OUT Parameter

```
DELIMITER //

DROP PROCEDURE IF EXISTS CountEmployees;

CREATE PROCEDURE CountEmployees(OUT total INT)
BEGIN
  SELECT COUNT(*) INTO total FROM employees;
END;
//

DELIMITER ;
```

```
-- Call and get result
CALL CountEmployees(@emp_count);
SELECT @emp_count AS total_employees;
```

Stored Procedure with INOUT Parameter

```
DELIMITER //

DROP PROCEDURE IF EXISTS AdjustSalary;

CREATE PROCEDURE AdjustSalary(INOUT emp_salary DECIMAL(10,2))
BEGIN
  SET emp_salary = emp_salary + (emp_salary * 0.10); -- Add 10%
END;
//

DELIMITER ;
```

```
-- Use variable
SET @mysalary = 50000;
CALL AdjustSalary(@mysalary);
SELECT @mysalary AS adjusted_salary;
```

Procedure with Logic (IF / CASE)

```
DELIMITER //

DROP PROCEDURE IF EXISTS SalaryGrade;

CREATE PROCEDURE SalaryGrade(IN emp_id_input INT)
BEGIN
  DECLARE emp_salary DECIMAL(10,2);

  SELECT salary INTO emp_salary FROM employees WHERE emp_id = emp_id_input;

  IF emp_salary < 50000 THEN
    SELECT 'Low' AS Grade;
  ELSEIF emp_salary < 70000 THEN
    SELECT 'Medium' AS Grade;
  ELSE
    SELECT 'High' AS Grade;
  END IF;
END;
//
DELIMITER ;
```

```
-- Call it:
CALL SalaryGrade(1);
```

Procedure with LOOP / WHILE / REPEAT (Basic Counter Example)

```
DELIMITER //

DROP PROCEDURE IF EXISTS PrintNumbers;

CREATE PROCEDURE PrintNumbers()
BEGIN
  DECLARE i INT DEFAULT 1;

  WHILE i <= 5 DO
    SELECT CONCAT('Number: ', i) AS output;
    SET i = i + 1;
  END WHILE;
END;
//

DELIMITER ;
```

```
CALL PrintNumbers();
```

Error Handling (DECLARE ... HANDLER)

```
DELIMITER //

DROP PROCEDURE IF EXISTS SafeSelect;

CREATE PROCEDURE SafeSelect()
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
  BEGIN
    SELECT 'An error occurred!' AS Error;
  END;

  -- Simulate error
  SELECT * FROM non_existing_table;
END;
//

DELIMITER ;
```

```
CALL SafeSelect();
```

Stored Procedures for CRUD operations

```
CREATE DATABASE IF NOT EXISTS crud_demo;
USE crud_demo;

-- Drop if exists
DROP TABLE IF EXISTS employees;

CREATE TABLE employees (
  emp_id INT AUTO_INCREMENT PRIMARY KEY,
  full_name VARCHAR(100),
  department VARCHAR(50),
  salary DECIMAL(10,2),
  hire_date DATE
);
```

CREATE (Insert a New Employee)

```
DELIMITER //

DROP PROCEDURE IF EXISTS CreateEmployee;

CREATE PROCEDURE CreateEmployee(
  IN p_full_name VARCHAR(100),
  IN p_department VARCHAR(50),
  IN p_salary DECIMAL(10,2),
  IN p_hire_date DATE
)
BEGIN
  INSERT INTO employees (full_name, department, salary, hire_date)
  VALUES (p_full_name, p_department, p_salary, p_hire_date);
END;
//

DELIMITER ;
```

```
CALL CreateEmployee('Alice Smith', 'HR', 45000, '2020-01-15');
```

READ (Get All Employees or by Department)

```
DELIMITER //

DROP PROCEDURE IF EXISTS GetEmployees;

CREATE PROCEDURE GetEmployees(IN p_department VARCHAR(50))
```

```
BEGIN
  IF p_department IS NULL OR p_department = '' THEN
    SELECT * FROM employees;
  ELSE
    SELECT * FROM employees WHERE department = p_department;
  END IF;
END;
//

DELIMITER ;
```

```
-- Example calls:
CALL GetEmployees('Finance');
CALL GetEmployees('');
```

UPDATE (Modify Employee Details)

```
DELIMITER //

DROP PROCEDURE IF EXISTS UpdateEmployee;

CREATE PROCEDURE UpdateEmployee(
  IN p_emp_id INT,
  IN p_full_name VARCHAR(100),
  IN p_department VARCHAR(50),
  IN p_salary DECIMAL(10,2),
  IN p_hire_date DATE
)
BEGIN
  UPDATE employees
  SET full_name = p_full_name,
    department = p_department,
    salary = p_salary,
    hire_date = p_hire_date
  WHERE emp_id = p_emp_id;
END;
//

DELIMITER ;
```

```
-- Example call:
CALL UpdateEmployee(1, 'Alice Smith', 'Finance', 48000, '2020-01-15');
```

DELETE (Remove an Employee)

```
DELIMITER //

DROP PROCEDURE IF EXISTS DeleteEmployee;

CREATE PROCEDURE DeleteEmployee(IN p_emp_id INT)
BEGIN
  DELETE FROM employees WHERE emp_id = p_emp_id;
END;
//

DELIMITER ;
```

```
-- Example call:
CALL DeleteEmployee(1);
```

Get Employee by ID

```
DELIMITER //

DROP PROCEDURE IF EXISTS GetEmployeeById;

CREATE PROCEDURE GetEmployeeById(IN p_emp_id INT)
BEGIN
  SELECT * FROM employees WHERE emp_id = p_emp_id;
END;
//

DELIMITER ;
```

```
-- Example call:
CALL GetEmployeeById(2);
```

SQL Triggers

Use Triggers to:

- Automatically perform actions before or after INSERT, UPDATE, DELETE
- Track changes (audit)
- Enforce rules or restrictions

```
CREATE DATABASE IF NOT EXISTS trigger_demo;
USE trigger_demo;

-- Drop tables if they already exist
DROP TABLE IF EXISTS employees_audit;
DROP TABLE IF EXISTS employees;

-- Main table
CREATE TABLE employees (
  emp_id INT AUTO_INCREMENT PRIMARY KEY,
  full_name VARCHAR(100),
  department VARCHAR(50),
  salary DECIMAL(10,2),
  hire_date DATE
);

-- Audit log table
CREATE TABLE employees_audit (
  audit_id INT AUTO_INCREMENT PRIMARY KEY,
  emp_id INT,
  action_type VARCHAR(10),
  action_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  old_salary DECIMAL(10,2),
  new_salary DECIMAL(10,2)
);
```

Create Triggers

BEFORE INSERT Trigger – Enforce minimum salary

```
DELIMITER //

DROP TRIGGER IF EXISTS before_insert_salary_check;

CREATE TRIGGER before_insert_salary_check
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
  IF NEW.salary < 30000 THEN
    SIGNAL SQLSTATE '45000'
```

```
    SET MESSAGE_TEXT = 'Salary must be at least 30,000';
  END IF;
END;
//

DELIMITER ;
```

AFTER INSERT Trigger – Log the insertion

```
DELIMITER //

DROP TRIGGER IF EXISTS after_insert_log;

CREATE TRIGGER after_insert_log
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
  INSERT INTO employees_audit (emp_id, action_type, new_salary)
  VALUES (NEW.emp_id, 'INSERT', NEW.salary);
END;
//

DELIMITER ;
```

BEFORE UPDATE Trigger – Track salary change in audit table

```
DELIMITER //

DROP TRIGGER IF EXISTS before_update_salary_log;

CREATE TRIGGER before_update_salary_log
BEFORE UPDATE ON employees
FOR EACH ROW
BEGIN
  IF NEW.salary <> OLD.salary THEN
    INSERT INTO employees_audit (emp_id, action_type, old_salary, new_salary)
    VALUES (OLD.emp_id, 'UPDATE', OLD.salary, NEW.salary);
  END IF;
END;
//

DELIMITER ;
```

AFTER DELETE Trigger – Log deletion

```
DELIMITER //

DROP TRIGGER IF EXISTS after_delete_log;

CREATE TRIGGER after_delete_log
AFTER DELETE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO employees_audit (emp_id, action_type, old_salary)
  VALUES (OLD.emp_id, 'DELETE', OLD.salary);
END;
//

DELIMITER ;
```

Test the Triggers

Insert with valid salary:

```
INSERT INTO employees (full_name, department, salary, hire_date)
VALUES ('Alice Smith', 'HR', 45000, '2020-01-15');
```

Insert with low salary:

```
-- Will throw an error
INSERT INTO employees (full_name, department, salary, hire_date)
VALUES ('Bob Lowpay', 'Finance', 25000, '2021-05-10');
```

Update salary to trigger log:

```
UPDATE employees SET salary = 50000 WHERE emp_id = 1;
```

Delete employee:

```
DELETE FROM employees WHERE emp_id = 1;
```

View Audit Logs:

```
SELECT * FROM employees_audit;
```