# Complete MySQL Notes - Employee Management System

## Table of Contents

---

## Introduction to MySQL

MySQL is a popular open-source relational database management system (RDBMS) that uses Structured Query Language (SQL) for managing data.

### Key Features:

- **Relational Database**: Data stored in tables with relationships
- **ACID Properties**: Atomicity, Consistency, Isolation, Durability
- **Cross-platform**: Works on Windows, Linux, macOS
- **Scalable**: Handles small to large applications
- **Open Source**: Free to use with commercial support available

---

## Database Design

### Employee Management System Schema

Our example will use an employee management system with the following entities:

```sql
-- Database Structure Overview
Departments (dept_id, dept_name, location, manager_id)
Employees (emp_id, first_name, last_name, email, phone, hire_date, salary, dept_id, manage
Projects (project_id, project_name, start_date, end_date, budget, dept_id)
Employee_Projects (emp_id, project_id, role, hours_worked)
Salaries (emp_id, salary, effective_date)
```

---

## DDL (Data Definition Language)

DDL commands are used to define and modify database structure.

### 1. CREATE DATABASE

```sql
-- Create the database
CREATE DATABASE employee_management;

-- Use the database
USE employee_management;

-- Show existing databases
SHOW DATABASES;
```

## 2. CREATE TABLE

```
sql
```

```sql
-- Create Departments table
CREATE TABLE departments (
    dept_id INT AUTO_INCREMENT PRIMARY KEY,
    dept_name VARCHAR(50) NOT NULL UNIQUE,
    location VARCHAR(100),
    manager_id INT,
    created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create Employees table
CREATE TABLE employees (
    emp_id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    phone VARCHAR(15),
    hire_date DATE NOT NULL,
    salary DECIMAL(10,2) NOT NULL,
    dept_id INT,
    manager_id INT,
    status ENUM('Active', 'Inactive', 'Terminated') DEFAULT 'Active',
    created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,

    -- Foreign key constraints
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id),
    FOREIGN KEY (manager_id) REFERENCES employees(emp_id)
);

-- Create Projects table
CREATE TABLE projects (
    project_id INT AUTO_INCREMENT PRIMARY KEY,
    project_name VARCHAR(100) NOT NULL,
    description TEXT,
    start_date DATE NOT NULL,
    end_date DATE,
    budget DECIMAL(12,2),
    status ENUM('Planning', 'Active', 'Completed', 'On Hold') DEFAULT 'Planning',
    dept_id INT,
    created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);

-- Create Employee_Projects junction table (Many-to-Many relationship)
CREATE TABLE employee_projects (
    emp_id INT,
    project_id INT,
    role VARCHAR(50),
    hours_worked DECIMAL(5,2) DEFAULT 0,
    assigned_date DATE DEFAULT (CURRENT_DATE),

    PRIMARY KEY (emp_id, project_id),
    FOREIGN KEY (emp_id) REFERENCES employees(emp_id) ON DELETE CASCADE,
    FOREIGN KEY (project_id) REFERENCES projects(project_id) ON DELETE CASCADE
);

-- Create Salary History table
CREATE TABLE salary_history (
    history_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_id INT,
    old_salary DECIMAL(10,2),
    new_salary DECIMAL(10,2),
    change_date DATE DEFAULT (CURRENT_DATE),
    reason VARCHAR(200),

    FOREIGN KEY (emp_id) REFERENCES employees(emp_id)
);
```

## 3. ALTER TABLE

```sql
-- Add new column
ALTER TABLE employees
ADD COLUMN birth_date DATE;

-- Modify column
ALTER TABLE employees
MODIFY COLUMN phone VARCHAR(20);

-- Drop column
ALTER TABLE employees
DROP COLUMN birth_date;

-- Add index
ALTER TABLE employees
ADD INDEX idx_email (email);

-- Add foreign key constraint
ALTER TABLE departments
ADD CONSTRAINT fk_dept_manager
FOREIGN KEY (manager_id) REFERENCES employees(emp_id);
```

## 4. DROP and TRUNCATE

```sql
-- Drop table (removes structure and data)
DROP TABLE IF EXISTS temp_table;

-- Truncate table (removes all data, keeps structure)
TRUNCATE TABLE salary_history;

-- Drop database
DROP DATABASE IF EXISTS old_database;
```

### 5. CREATE INDEX

```sql
-- Create single column index
CREATE INDEX idx_last_name ON employees(last_name);

-- Create composite index
CREATE INDEX idx_dept_salary ON employees(dept_id, salary);

-- Create unique index
CREATE UNIQUE INDEX idx_emp_email ON employees(email);

-- Show indexes
SHOW INDEX FROM employees;
```

---

## DML (Data Manipulation Language)

DML commands are used to manipulate data within tables.

# 1. INSERT

sql

```sql
-- Insert into departments
INSERT INTO departments (dept_name, location) VALUES
('Human Resources', 'New York'),
('Information Technology', 'San Francisco'),
('Finance', 'Chicago'),
('Marketing', 'Los Angeles'),
('Operations', 'Denver');

-- Insert into employees
INSERT INTO employees (first_name, last_name, email, phone, hire_date, salary, dept_id) VA
('John', 'Smith', 'john.smith@company.com', '555-0101', '2023-01-15', 75000.00, 2),
('Sarah', 'Johnson', 'sarah.johnson@company.com', '555-0102', '2023-02-01', 85000.00, 2),
('Michael', 'Brown', 'michael.brown@company.com', '555-0103', '2023-01-20', 65000.00, 1),
('Emily', 'Davis', 'emily.davis@company.com', '555-0104', '2023-03-10', 70000.00, 3),
('David', 'Wilson', 'david.wilson@company.com', '555-0105', '2023-02-15', 90000.00, 2);

-- Update manager_id after employees are inserted
UPDATE departments SET manager_id = 3 WHERE dept_id = 1; -- Michael manages HR
UPDATE departments SET manager_id = 5 WHERE dept_id = 2; -- David manages IT

-- Insert projects
INSERT INTO projects (project_name, description, start_date, end_date, budget, dept_id) VA
('Employee Portal', 'Internal employee management system', '2024-01-01', '2024-06-30', 15
('Payroll System', 'Automated payroll processing', '2024-02-01', '2024-08-31', 200000.00,
('Marketing Campaign', 'Q2 product launch campaign', '2024-03-01', '2024-05-31', 75000.00

-- Insert employee-project assignments
INSERT INTO employee_projects (emp_id, project_id, role, hours_worked) VALUES
(1, 1, 'Developer', 120.5),
(2, 1, 'Senior Developer', 150.0),
(5, 1, 'Project Manager', 80.0),
(4, 2, 'Financial Analyst', 100.0),
(2, 3, 'Technical Consultant', 40.0);
```

## 2. UPDATE

sql

```sql
-- Simple update
UPDATE employees
SET salary = 80000.00
WHERE emp_id = 1;

-- Update with calculation
UPDATE employees
SET salary = salary * 1.05
WHERE dept_id = 2;

-- Update with JOIN
UPDATE employees e
JOIN departments d ON e.dept_id = d.dept_id
SET e.salary = e.salary * 1.03
WHERE d.dept_name = 'Finance';

-- Update multiple columns
UPDATE employees
SET salary = 95000.00, status = 'Active'
WHERE emp_id = 5;

-- Record salary change in history
INSERT INTO salary_history (emp_id, old_salary, new_salary, reason)
SELECT emp_id, 75000.00, salary, 'Annual Review'
FROM employees
WHERE emp_id = 1;
```

### 3. DELETE

```sql
-- Delete specific record
DELETE FROM employee_projects
WHERE emp_id = 2 AND project_id = 3;

-- Delete with condition
DELETE FROM employees
WHERE status = 'Terminated' AND hire_date < '2020-01-01';

-- Delete with subquery
DELETE FROM salary_history
WHERE emp_id NOT IN (SELECT emp_id FROM employees);

-- Safe delete with EXISTS
DELETE e FROM employees e
WHERE EXISTS (
    SELECT 1 FROM employees
    WHERE manager_id = e.emp_id
    AND status = 'Terminated'
);
```

---

## DQL (Data Query Language)

DQL is used to retrieve data from the database.

### 1. Basic SELECT

sql

```
-- Select all columns
SELECT * FROM employees;

-- Select specific columns
SELECT first_name, last_name, email, salary FROM employees;

-- Select with alias
SELECT
    first_name AS 'First Name',
    last_name AS 'Last Name',
    salary AS 'Annual Salary'
FROM employees;

-- Select with expressions
SELECT
    first_name,
    last_name,
    salary,
    salary * 12 AS annual_salary,
    YEAR(hire_date) AS hire_year
FROM employees;
```

## 2. WHERE Clause

sql

```sql
-- Basic conditions
SELECT * FROM employees WHERE salary > 70000;

-- Multiple conditions
SELECT * FROM employees
WHERE dept_id = 2 AND salary BETWEEN 70000 AND 90000;

-- Pattern matching
SELECT * FROM employees
WHERE first_name LIKE 'J%' OR last_name LIKE '%son';

-- IN operator
SELECT * FROM employees
WHERE dept_id IN (1, 2, 3);

-- NULL checks
SELECT * FROM projects
WHERE end_date IS NULL;

-- Date conditions
SELECT * FROM employees
WHERE hire_date >= '2023-01-01' AND hire_date < '2024-01-01';
```

### 3. ORDER BY and LIMIT

sql

```sql
-- Single column sorting
SELECT * FROM employees ORDER BY salary DESC;

-- Multiple column sorting
SELECT * FROM employees
ORDER BY dept_id ASC, salary DESC;

-- Limit results
SELECT * FROM employees
ORDER BY salary DESC
LIMIT 5;

-- Pagination
SELECT * FROM employees
ORDER BY emp_id
LIMIT 10 OFFSET 20;
```

## 4. Aggregate Functions

sql

```sql
-- Count employees
SELECT COUNT(*) AS total_employees FROM employees;

-- Count by department
SELECT
    d.dept_name,
    COUNT(e.emp_id) AS employee_count
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id
GROUP BY d.dept_id, d.dept_name;

-- Salary statistics
SELECT
    AVG(salary) AS avg_salary,
    MIN(salary) AS min_salary,
    MAX(salary) AS max_salary,
    SUM(salary) AS total_payroll
FROM employees;

-- Department-wise statistics
SELECT
    d.dept_name,
    COUNT(e.emp_id) AS emp_count,
    AVG(e.salary) AS avg_salary,
    SUM(e.salary) AS total_payroll
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id
GROUP BY d.dept_id, d.dept_name
HAVING COUNT(e.emp_id) > 0
ORDER BY avg_salary DESC;
```

## 5. GROUP BY and HAVING

sql

```sql
-- Group by single column
SELECT dept_id, COUNT(*) as emp_count, AVG(salary) as avg_salary
FROM employees
GROUP BY dept_id;

-- Group by multiple columns
SELECT
    YEAR(hire_date) AS hire_year,
    dept_id,
    COUNT(*) AS new_hires
FROM employees
GROUP BY YEAR(hire_date), dept_id
ORDER BY hire_year, dept_id;

-- HAVING clause
SELECT
    dept_id,
    AVG(salary) AS avg_salary
FROM employees
GROUP BY dept_id
HAVING AVG(salary) > 70000;
```

## 6. JOINS

```
sql
```

```sql
-- INNER JOIN
SELECT
    e.first_name,
    e.last_name,
    d.dept_name,
    e.salary
FROM employees e
INNER JOIN departments d ON e.dept_id = d.dept_id;

-- LEFT JOIN
SELECT
    d.dept_name,
    e.first_name,
    e.last_name
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id;

-- RIGHT JOIN
SELECT
    e.first_name,
    e.last_name,
    d.dept_name
FROM employees e
RIGHT JOIN departments d ON e.dept_id = d.dept_id;

-- FULL OUTER JOIN (MySQL doesn't support directly, use UNION)
SELECT d.dept_name, e.first_name, e.last_name
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id
UNION
SELECT d.dept_name, e.first_name, e.last_name
FROM departments d
```

```sql
RIGHT JOIN employees e ON d.dept_id = e.dept_id;

-- Self JOIN (Employee and Manager)
SELECT
    e.first_name AS employee_name,
    m.first_name AS manager_name
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.emp_id;

-- Multiple JOINs
SELECT
    e.first_name,
    e.last_name,
    d.dept_name,
    p.project_name,
    ep.role,
    ep.hours_worked
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
JOIN employee_projects ep ON e.emp_id = ep.emp_id
JOIN projects p ON ep.project_id = p.project_id;
```

## 7. Subqueries

```
sql
```

```sql
-- Scalar subquery
SELECT first_name, last_name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);

-- Correlated subquery
SELECT e1.first_name, e1.last_name, e1.salary, e1.dept_id
FROM employees e1
WHERE e1.salary > (
    SELECT AVG(e2.salary)
    FROM employees e2
    WHERE e2.dept_id = e1.dept_id
);

-- EXISTS subquery
SELECT d.dept_name
FROM departments d
WHERE EXISTS (
    SELECT 1 FROM employees e
    WHERE e.dept_id = d.dept_id
);

-- IN subquery
SELECT first_name, last_name
FROM employees
WHERE dept_id IN (
    SELECT dept_id FROM departments
    WHERE location = 'San Francisco'
);

-- Subquery in FROM clause
SELECT dept_summary.dept_name, dept_summary.avg_salary
```

```
FROM (
    SELECT d.dept_name, AVG(e.salary) as avg_salary
    FROM departments d
    JOIN employees e ON d.dept_id = e.dept_id
    GROUP BY d.dept_id, d.dept_name
) AS dept_summary
WHERE dept_summary.avg_salary > 75000;
```

---

## Advanced Concepts

### 1. Views

```
sql
```

```sql
-- Create a view for employee details
CREATE VIEW employee_details AS
SELECT
    e.emp_id,
    CONCAT(e.first_name, ' ', e.last_name) AS full_name,
    e.email,
    e.salary,
    d.dept_name,
    CONCAT(m.first_name, ' ', m.last_name) AS manager_name
FROM employees e
LEFT JOIN departments d ON e.dept_id = d.dept_id
LEFT JOIN employees m ON e.manager_id = m.emp_id
WHERE e.status = 'Active';

-- Use the view
SELECT * FROM employee_details WHERE salary > 70000;

-- Create view for department summary
CREATE VIEW department_summary AS
SELECT
    d.dept_name,
    d.location,
    COUNT(e.emp_id) AS employee_count,
    AVG(e.salary) AS avg_salary,
    SUM(e.salary) AS total_payroll
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id AND e.status = 'Active'
GROUP BY d.dept_id, d.dept_name, d.location;

-- Drop view
DROP VIEW IF EXISTS employee_details;
```

## 2. Stored Procedures

```
sql
```

```sql
-- Simple stored procedure
DELIMITER //
CREATE PROCEDURE GetEmployeesByDepartment(IN dept_name VARCHAR(50))
BEGIN
    SELECT e.first_name, e.last_name, e.salary
    FROM employees e
    JOIN departments d ON e.dept_id = d.dept_id
    WHERE d.dept_name = dept_name AND e.status = 'Active';
END //
DELIMITER ;

-- Call the procedure
CALL GetEmployeesByDepartment('Information Technology');

-- Procedure with OUT parameter
DELIMITER //
CREATE PROCEDURE GetDepartmentStats(
    IN dept_name VARCHAR(50),
    OUT emp_count INT,
    OUT avg_salary DECIMAL(10,2)
)
BEGIN
    SELECT COUNT(*), AVG(salary)
    INTO emp_count, avg_salary
    FROM employees e
    JOIN departments d ON e.dept_id = d.dept_id
    WHERE d.dept_name = dept_name AND e.status = 'Active';
END //
DELIMITER ;

-- Call procedure with OUT parameters
CALL GetDepartmentStats('Information Technology', @count, @avg_sal);
```

```sql
SELECT @count AS employee_count, @avg_sal AS average_salary;

-- Procedure for salary increase
DELIMITER //
CREATE PROCEDURE GiveSalaryIncrease(
    IN emp_id INT,
    IN increase_percent DECIMAL(5,2),
    OUT new_salary DECIMAL(10,2)
)
BEGIN
    DECLARE old_salary DECIMAL(10,2);

    -- Get current salary
    SELECT salary INTO old_salary FROM employees WHERE employees.emp_id = emp_id;

    -- Calculate new salary
    SET new_salary = old_salary * (1 + increase_percent / 100);

    -- Update employee salary
    UPDATE employees SET salary = new_salary WHERE employees.emp_id = emp_id;

    -- Log the change
    INSERT INTO salary_history (emp_id, old_salary, new_salary, reason)
    VALUES (emp_id, old_salary, new_salary, CONCAT('Salary increase of ', increase_percent
END //
DELIMITER ;
```

### 3. Functions

```
sql
```

```sql
-- Create a function to calculate experience
DELIMITER //
CREATE FUNCTION CalculateExperience(hire_date DATE)
RETURNS DECIMAL(4,2)
READS SQL DATA
DETERMINISTIC
BEGIN
    DECLARE experience DECIMAL(4,2);
    SET experience = TIMESTAMPDIFF(MONTH, hire_date, CURDATE()) / 12;
    RETURN experience;
END //
DELIMITER ;

-- Use the function
SELECT
    first_name,
    last_name,
    hire_date,
    CalculateExperience(hire_date) AS years_experience
FROM employees;

-- Function to get employee grade based on salary
DELIMITER //
CREATE FUNCTION GetEmployeeGrade(salary DECIMAL(10,2))
RETURNS VARCHAR(10)
DETERMINISTIC
BEGIN
    DECLARE grade VARCHAR(10);

    IF salary >= 90000 THEN
        SET grade = 'Senior';
    ELSEIF salary >= 70000 THEN
```

```
        SET grade = 'Mid-level';
    ELSE
        SET grade = 'Junior';
    END IF;

    RETURN grade;
END //
DELIMITER ;

-- Use the grade function
SELECT
    first_name,
    last_name,
    salary,
    GetEmployeeGrade(salary) AS employee_grade
FROM employees;
```

## 4. Triggers

```
sql
```

```sql
-- Trigger to automatically update salary history
DELIMITER //
CREATE TRIGGER salary_change_trigger
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
    IF OLD.salary != NEW.salary THEN
        INSERT INTO salary_history (emp_id, old_salary, new_salary, reason)
        VALUES (NEW.emp_id, OLD.salary, NEW.salary, 'Salary Updated');
    END IF;
END //
DELIMITER ;

-- Trigger to prevent deletion of employees with active projects
DELIMITER //
CREATE TRIGGER prevent_employee_deletion
BEFORE DELETE ON employees
FOR EACH ROW
BEGIN
    DECLARE project_count INT;

    SELECT COUNT(*) INTO project_count
    FROM employee_projects ep
    JOIN projects p ON ep.project_id = p.project_id
    WHERE ep.emp_id = OLD.emp_id AND p.status = 'Active';

    IF project_count > 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Cannot delete employee with active projects';
    END IF;
END //
DELIMITER ;
```

```sql
-- Trigger to update project budget when employee hours change
DELIMITER //
CREATE TRIGGER update_project_hours
AFTER UPDATE ON employee_projects
FOR EACH ROW
BEGIN
    DECLARE total_hours DECIMAL(8,2);

    IF OLD.hours_worked != NEW.hours_worked THEN
        SELECT SUM(hours_worked) INTO total_hours
        FROM employee_projects
        WHERE project_id = NEW.project_id;

        -- You could add logic here to update project status or costs
        -- based on total hours worked
    END IF;
END //
DELIMITER ;
```

## 5. Transactions

```
sql
```

```sql
-- Basic transaction example
START TRANSACTION;

INSERT INTO employees (first_name, last_name, email, hire_date, salary, dept_id)
VALUES ('Alice', 'Cooper', 'alice.cooper@company.com', '2024-01-15', 72000.00, 2);

SET @new_emp_id = LAST_INSERT_ID();

INSERT INTO employee_projects (emp_id, project_id, role)
VALUES (@new_emp_id, 1, 'Developer');

COMMIT;

-- Transaction with rollback
START TRANSACTION;

UPDATE employees SET salary = salary * 1.10 WHERE dept_id = 2;

-- Check if update is acceptable
SELECT COUNT(*) as high_salary_count
FROM employees
WHERE salary > 100000 AND dept_id = 2;

-- If too many high salaries, rollback
ROLLBACK;

-- Transaction with savepoints
START TRANSACTION;

SAVEPOINT before_salary_update;

UPDATE employees SET salary = salary * 1.05 WHERE dept_id = 1;
```

```
SAVEPOINT after_hr_update;

UPDATE employees SET salary = salary * 1.08 WHERE dept_id = 2;

-- If something goes wrong, can rollback to savepoint
ROLLBACK TO before_salary_update;

COMMIT;
```

## 6. Common Table Expressions (CTEs)

```
sql
```

```sql
-- Simple CTE
WITH department_avg AS (
    SELECT dept_id, AVG(salary) as avg_salary
    FROM employees
    GROUP BY dept_id
)
SELECT e.first_name, e.last_name, e.salary, da.avg_salary
FROM employees e
JOIN department_avg da ON e.dept_id = da.dept_id
WHERE e.salary > da.avg_salary;

-- Recursive CTE for organizational hierarchy
WITH RECURSIVE employee_hierarchy AS (
    -- Base case: top-level employees (no manager)
    SELECT emp_id, first_name, last_name, manager_id, 0 as level
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive case: employees with managers
    SELECT e.emp_id, e.first_name, e.last_name, e.manager_id, eh.level + 1
    FROM employees e
    JOIN employee_hierarchy eh ON e.manager_id = eh.emp_id
)
SELECT emp_id,
       CONCAT(REPEAT('  ', level), first_name, ' ', last_name) as employee_hierarchy,
       level
FROM employee_hierarchy
ORDER BY level, last_name;

-- Multiple CTEs
```

```sql
WITH
high_performers AS (
    SELECT emp_id, first_name, last_name, salary
    FROM employees
    WHERE salary > (SELECT AVG(salary) FROM employees)
),
project_leaders AS (
    SELECT DISTINCT ep.emp_id
    FROM employee_projects ep
    WHERE ep.role LIKE '%Manager%' OR ep.role LIKE '%Lead%'
)
SELECT hp.first_name, hp.last_name, hp.salary
FROM high_performers hp
JOIN project_leaders pl ON hp.emp_id = pl.emp_id;
```

## Database Security

### 1. User Management

sql

```
-- Create users
CREATE USER 'hr_manager'@'localhost' IDENTIFIED BY 'secure_password';
CREATE USER 'developer'@'%' IDENTIFIED BY 'dev_password';
CREATE USER 'readonly_user'@'localhost' IDENTIFIED BY 'read_password';

-- Grant privileges
GRANT ALL PRIVILEGES ON employee_management.* TO 'hr_manager'@'localhost';
GRANT SELECT, INSERT, UPDATE ON employee_management.employees TO 'developer'@'%';
GRANT SELECT ON employee_management.* TO 'readonly_user'@'localhost';

-- Grant specific column privileges
GRANT SELECT (first_name, last_name, email) ON employees TO 'limited_user'@'localhost';

-- Revoke privileges
REVOKE INSERT ON employee_management.employees FROM 'developer'@'%';

-- Show privileges
SHOW GRANTS FOR 'hr_manager'@'localhost';

-- Drop user
DROP USER 'old_user'@'localhost';
```

## 2. Data Validation and Constraints

```sql
-- Add check constraints (MySQL 8.0+)
ALTER TABLE employees
ADD CONSTRAINT chk_salary CHECK (salary > 0);

ALTER TABLE employees
ADD CONSTRAINT chk_hire_date CHECK (hire_date <= CURDATE());

-- Create table with constraints
CREATE TABLE employee_reviews (
    review_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_id INT,
    review_date DATE DEFAULT (CURDATE()),
    rating DECIMAL(3,2),
    comments TEXT,

    CONSTRAINT fk_review_emp FOREIGN KEY (emp_id) REFERENCES employees(emp_id),
    CONSTRAINT chk_rating CHECK (rating >= 1.0 AND rating <= 5.0),
    CONSTRAINT chk_review_date CHECK (review_date <= CURDATE())
);
```

---

## Performance Optimization

### 1. Indexing Strategies

sql

```sql
-- Analyze query performance
EXPLAIN SELECT * FROM employees WHERE last_name = 'Smith';

-- Create covering index
CREATE INDEX idx_emp_dept_salary ON employees(dept_id, salary, first_name, last_name);

-- Create partial index (MySQL doesn't support, but here's the concept)
-- CREATE INDEX idx_active_employees ON employees(dept_id) WHERE status = 'Active';

-- Monitor index usage
SELECT
    table_name,
    index_name,
    column_name,
    cardinality
FROM information_schema.statistics
WHERE table_schema = 'employee_management'
ORDER BY table_name, index_name;
```

## 2. Query Optimization

```sql
-- Inefficient query
SELECT * FROM employees e1
WHERE salary > (
    SELECT AVG(salary) FROM employees e2 WHERE e2.dept_id = e1.dept_id
);

-- Optimized version using JOIN
SELECT e.*, dept_avg.avg_salary
FROM employees e
JOIN (
    SELECT dept_id, AVG(salary) as avg_salary
    FROM employees
    GROUP BY dept_id
) dept_avg ON e.dept_id = dept_avg.dept_id
WHERE e.salary > dept_avg.avg_salary;

-- Use LIMIT for large result sets
SELECT * FROM employees
ORDER BY hire_date DESC
LIMIT 10;

-- Avoid SELECT * when possible
SELECT emp_id, first_name, last_name, salary
FROM employees
WHERE dept_id = 2;
```

## 3. Database Maintenance

sql

```sql
-- Analyze table statistics
ANALYZE TABLE employees;

-- Optimize table
OPTIMIZE TABLE employees;

-- Check table integrity
CHECK TABLE employees;

-- Repair table if needed
REPAIR TABLE employees;

-- Show table status
SHOW TABLE STATUS FROM employee_management;
```

---

## Practical Examples and Use Cases

### 1. Employee Reporting Queries

sql

```
-- Monthly hiring report
SELECT
    DATE_FORMAT(hire_date, '%Y-%m') as hire_month,
    COUNT(*) as new_hires,
    AVG(salary) as avg_starting_salary
FROM employees
WHERE hire_date >= DATE_SUB(CURDATE(), INTERVAL 12 MONTH)
GROUP BY DATE_FORMAT(hire_date, '%Y-%m')
ORDER BY hire_month;

-- Department performance dashboard
SELECT
    d.dept_name,
    COUNT(e.emp_id) as employee_count,
    AVG(e.salary) as avg_salary,
    SUM(CASE WHEN e.hire_date >= DATE_SUB(CURDATE(), INTERVAL 1 YEAR) THEN 1 ELSE 0 END) a
    COUNT(p.project_id) as active_projects
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id AND e.status = 'Active'
LEFT JOIN projects p ON d.dept_id = p.dept_id AND p.status = 'Active'
GROUP BY d.dept_id, d.dept_name
ORDER BY employee_count DESC;

-- Employee project workload
SELECT
    e.first_name,
    e.last_name,
    COUNT(ep.project_id) as project_count,
    SUM(ep.hours_worked) as total_hours,
    AVG(ep.hours_worked) as avg_hours_per_project
FROM employees e
JOIN employee_projects ep ON e.emp_id = ep.emp_id
```

```sql
JOIN projects p ON ep.project_id = p.project_id
WHERE p.status IN ('Active', 'Planning')
GROUP BY e.emp_id, e.first_name, e.last_name
HAVING project_count > 1
ORDER BY total_hours DESC;
```

## 2. Data Migration and Cleanup

```sql
-- Archive old salary history
CREATE TABLE salary_history_archive AS
SELECT * FROM salary_history
WHERE change_date < DATE_SUB(CURDATE(), INTERVAL 2 YEAR);

DELETE FROM salary_history
WHERE change_date < DATE_SUB(CURDATE(), INTERVAL 2 YEAR);

-- Update employee status based on project activity
UPDATE employees
SET status = 'Inactive'
WHERE emp_id NOT IN (
    SELECT DISTINCT ep.emp_id
    FROM employee_projects ep
    JOIN projects p ON ep.project_id = p.project_id
    WHERE p.status = 'Active'
)
AND status = 'Active'
AND hire_date < DATE_SUB(CURDATE(), INTERVAL 6 MONTH);

-- Clean up duplicate email addresses
DELETE e1 FROM employees e1
INNER JOIN employees e2
WHERE e1.emp_id < e2.emp_id
AND e1.email = e2.email;
```

## 3. Advanced Analytics Queries

sql

```sql
-- Salary distribution analysis
SELECT
    CASE
        WHEN salary < 50000 THEN 'Under 50K'
        WHEN salary BETWEEN 50000 AND 70000 THEN '50K-70K'
        WHEN salary BETWEEN 70001 AND 90000 THEN '70K-90K'
        WHEN salary > 90000 THEN 'Over 90K'
    END AS salary_range,
    COUNT(*) as employee_count,
    ROUND(COUNT(*) * 100.0 / (SELECT COUNT(*) FROM employees), 2) as percentage
FROM employees
WHERE status = 'Active'
GROUP BY salary_range
ORDER BY MIN(salary);

-- Employee retention analysis
SELECT
    d.dept_name,
    COUNT(e.emp_id) as total_employees,
    SUM(CASE WHEN e.hire_date >= DATE_SUB(CURDATE(), INTERVAL 1 YEAR) THEN 1 ELSE 0 END) a
    SUM(CASE WHEN e.status = 'Terminated' AND e.updated_date >= DATE_SUB(CURDATE(), INTERV
    ROUND(
        (COUNT(e.emp_id) - SUM(CASE WHEN e.status = 'Terminated' AND e.updated_date >= DAT
        2
    ) as retention_rate
FROM departments d
LEFT JOIN employees e ON d.dept_id = e.dept_id
GROUP BY d.dept_id, d.dept_name
ORDER BY retention_rate DESC;

-- Project timeline and resource allocation
SELECT
```

```sql
    p.project_name,
    p.start_date,
    p.end_date,
    DATEDIFF(p.end_date, p.start_date) as duration_days,
    COUNT(ep.emp_id) as team_size,
    SUM(ep.hours_worked) as total_hours,
    p.budget,
    ROUND(p.budget / NULLIF(SUM(ep.hours_worked), 0), 2) as cost_per_hour
FROM projects p
LEFT JOIN employee_projects ep ON p.project_id = ep.project_id
WHERE p.status IN ('Active', 'Completed')
GROUP BY p.project_id, p.project_name, p.start_date, p.end_date, p.budget
ORDER BY p.start_date DESC;
```

---

## Window Functions (MySQL 8.0+)

Window functions perform calculations across a set of table rows related to the current row.

### 1. Ranking Functions

```
sql
```

```sql
-- Rank employees by salary within each department
SELECT
    first_name,
    last_name,
    dept_id,
    salary,
    RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) as salary_rank,
    DENSE_RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) as dense_rank,
    ROW_NUMBER() OVER (PARTITION BY dept_id ORDER BY salary DESC) as row_num
FROM employees
WHERE status = 'Active';

-- Top 3 highest paid employees per department
SELECT *
FROM (
    SELECT
        e.first_name,
        e.last_name,
        d.dept_name,
        e.salary,
        RANK() OVER (PARTITION BY e.dept_id ORDER BY e.salary DESC) as salary_rank
    FROM employees e
    JOIN departments d ON e.dept_id = d.dept_id
    WHERE e.status = 'Active'
) ranked_employees
WHERE salary_rank <= 3;

-- Percentile ranking
SELECT
    first_name,
    last_name,
    salary,
    PERCENT_RANK() OVER (ORDER BY salary) as percentile_rank,
    CUME_DIST() OVER (ORDER BY salary) as cumulative_distribution
FROM employees
WHERE status = 'Active';
```

## 2. Aggregate Window Functions

sql

```sql
-- Running total of salaries
SELECT
    first_name,
    last_name,
    hire_date,
    salary,
    SUM(salary) OVER (ORDER BY hire_date) as running_total_payroll
FROM employees
WHERE status = 'Active'
ORDER BY hire_date;

-- Moving average salary (3-employee window)
SELECT
    first_name,
    last_name,
    hire_date,
    salary,
    AVG(salary) OVER (
        ORDER BY hire_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) as moving_avg_salary
FROM employees
WHERE status = 'Active'
ORDER BY hire_date;

-- Department salary statistics with individual comparison
SELECT
    e.first_name,
    e.last_name,
    d.dept_name,
    e.salary,
    AVG(e.salary) OVER (PARTITION BY e.dept_id) as dept_avg_salary,
    MAX(e.salary) OVER (PARTITION BY e.dept_id) as dept_max_salary,
    MIN(e.salary) OVER (PARTITION BY e.dept_id) as dept_min_salary,
    e.salary - AVG(e.salary) OVER (PARTITION BY e.dept_id) as salary_vs_dept_avg
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE e.status = 'Active';
```

## 3. Value Functions

```
sql
```

```sql
-- Compare with previous and next employee salaries
SELECT
    first_name,
    last_name,
    hire_date,
    salary,
    LAG(salary, 1) OVER (ORDER BY hire_date) as prev_hire_salary,
    LEAD(salary, 1) OVER (ORDER BY hire_date) as next_hire_salary,
    salary - LAG(salary, 1) OVER (ORDER BY hire_date) as salary_diff_from_prev
FROM employees
WHERE status = 'Active'
ORDER BY hire_date;

-- First and last values in each department
SELECT
    first_name,
    last_name,
    dept_id,
    hire_date,
    salary,
    FIRST_VALUE(salary) OVER (
        PARTITION BY dept_id
        ORDER BY hire_date
        ROWS UNBOUNDED PRECEDING
    ) as first_dept_salary,
    LAST_VALUE(salary) OVER (
        PARTITION BY dept_id
        ORDER BY hire_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) as last_dept_salary
```

```
FROM employees
WHERE status = 'Active';
```

---

## JSON Data Type and Functions (MySQL 5.7+)

MySQL supports JSON data type for storing and manipulating JSON documents.

### 1. JSON Column and Basic Operations

```
sql
```

```sql
-- Add JSON column to employees table
ALTER TABLE employees ADD COLUMN skills JSON;
ALTER TABLE employees ADD COLUMN contact_info JSON;

-- Insert JSON data
UPDATE employees
SET skills = JSON_ARRAY('MySQL', 'Python', 'JavaScript'),
    contact_info = JSON_OBJECT(
        'emergency_contact', JSON_OBJECT('name', 'Jane Smith', 'phone', '555-9999'),
        'address', JSON_OBJECT('street', '123 Main St', 'city', 'New York', 'zip', '10001
    )
WHERE emp_id = 1;

UPDATE employees
SET skills = JSON_ARRAY('Java', 'Spring', 'React'),
    contact_info = JSON_OBJECT(
        'emergency_contact', JSON_OBJECT('name', 'Bob Johnson', 'phone', '555-8888'),
        'address', JSON_OBJECT('street', '456 Oak Ave', 'city', 'San Francisco', 'zip', '9
    )
WHERE emp_id = 2;

-- Query JSON data
SELECT
    first_name,
    last_name,
    JSON_EXTRACT(skills, '$[0]') as primary_skill,
    JSON_EXTRACT(contact_info, '$.emergency_contact.name') as emergency_contact
FROM employees
WHERE skills IS NOT NULL;

-- Using -> and ->> operators (shorthand for JSON_EXTRACT)
SELECT
```

```
    first_name,
    last_name,
    skills->'$[0]' as primary_skill,
    contact_info->>'$.address.city' as city
FROM employees
WHERE skills IS NOT NULL;
```

## 2. JSON Functions

```
sql
```

```sql
-- JSON search and manipulation
SELECT
    first_name,
    last_name,
    skills,
    JSON_LENGTH(skills) as skill_count,
    JSON_CONTAINS(skills, '"MySQL"') as knows_mysql,
    JSON_SEARCH(skills, 'one', 'Python') as python_position
FROM employees
WHERE skills IS NOT NULL;

-- JSON array operations
SELECT
    first_name,
    last_name,
    skills,
    JSON_ARRAY_APPEND(skills, ', 'Docker') as skills_with_docker,
    JSON_ARRAY_INSERT(skills, '$[1]', 'Git') as skills_with_git
FROM employees
WHERE emp_id = 1;

-- JSON object operations
SELECT
    first_name,
    last_name,
    contact_info,
    JSON_SET(contact_info, '$.phone', '555-1234') as updated_contact,
    JSON_REMOVE(contact_info, '$.address.zip') as contact_without_zip
FROM employees
WHERE contact_info IS NOT NULL
LIMIT 1;
```

### 3. JSON Indexing

```sql
-- Create functional index on JSON column
ALTER TABLE employees
ADD INDEX idx_primary_skill ((CAST(skills->'$[0]' AS CHAR(50))));

-- Create index on JSON path
ALTER TABLE employees
ADD INDEX idx_city ((CAST(contact_info->>'$.address.city' AS CHAR(50))));

-- Query using JSON index
SELECT first_name, last_name, skills
FROM employees
WHERE skills->'$[0]' = 'MySQL';
```

---

## Backup and Recovery

### 1. Logical Backup with mysqldump

```bash
# Complete database backup
mysqldump -u root -p employee_management > employee_management_backup.sql

# Backup specific tables
mysqldump -u root -p employee_management employees departments > partial_backup.sql

# Backup with additional options
mysqldump -u root -p \
  --single-transaction \
  --routines \
  --triggers \
  --events \
  employee_management > complete_backup.sql

# Backup all databases
mysqldump -u root -p --all-databases > all_databases_backup.sql
```

## 2. Point-in-Time Recovery

```sql
-- Enable binary logging (in my.cnf)
-- log-bin=mysql-bin
-- binlog-format=ROW

-- Show binary logs
SHOW BINARY LOGS;

-- Show binary log events
SHOW BINLOG EVENTS IN 'mysql-bin.000001';

-- Create a backup point
FLUSH LOGS;
```

### 3. Database Restoration

```bash
# Restore from backup
mysql -u root -p employee_management < employee_management_backup.sql

# Restore specific tables
mysql -u root -p employee_management < partial_backup.sql

# Create database before restore
mysql -u root -p -e "CREATE DATABASE employee_management_restored;"
mysql -u root -p employee_management_restored < employee_management_backup.sql
```

---

## Monitoring and Troubleshooting

## 1. Performance Monitoring

```sql
-- Show running processes
SHOW PROCESSLIST;

-- Show slow queries (enable slow query log first)
-- In my.cnf: slow_query_log = 1, long_query_time = 2

-- Check table locks
SHOW OPEN TABLES WHERE In_use > 0;

-- Show engine status
SHOW ENGINE INNODB STATUS;

-- Query cache statistics
SHOW STATUS LIKE 'Qcache%';

-- Connection statistics
SHOW STATUS LIKE 'Connections';
SHOW STATUS LIKE 'Threads_connected';
SHOW STATUS LIKE 'Max_used_connections';
```

## 2. Space Usage Analysis

sql

```sql
-- Database size
SELECT
    table_schema AS 'Database',
    ROUND(SUM(data_length + index_length) / 1024 / 1024, 2) AS 'Size (MB)'
FROM information_schema.tables
WHERE table_schema = 'employee_management'
GROUP BY table_schema;

-- Table sizes
SELECT
    table_name AS 'Table',
    ROUND(((data_length + index_length) / 1024 / 1024), 2) AS 'Size (MB)',
    table_rows AS 'Rows'
FROM information_schema.tables
WHERE table_schema = 'employee_management'
ORDER BY (data_length + index_length) DESC;

-- Index usage
SELECT
    t.table_name,
    t.index_name,
    t.column_name,
    s.cardinality,
    ROUND(s.cardinality / tr.table_rows * 100, 2) as selectivity_percent
FROM information_schema.statistics s
JOIN information_schema.tables tr ON s.table_name = tr.table_name
JOIN information_schema.statistics t ON s.table_name = t.table_name
WHERE s.table_schema = 'employee_management'
AND tr.table_schema = 'employee_management'
ORDER BY selectivity_percent DESC;
```

## Best Practices and Tips

### 1. Database Design Best Practices

sql

```
-- Use appropriate data types
-- Good
CREATE TABLE example_good (
    id INT AUTO_INCREMENT PRIMARY KEY,
    status ENUM('active', 'inactive') NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    amount DECIMAL(10,2) NOT NULL
);

-- Avoid
CREATE TABLE example_bad (
    id VARCHAR(50) PRIMARY KEY,  -- Use INT for IDs
    status VARCHAR(255),         -- Use ENUM for limited options
    created_at VARCHAR(50),      -- Use proper date/time types
    amount FLOAT                 -- Use DECIMAL for money
);

-- Normalize your database
-- Instead of storing repeated department info in employees table
-- employees (emp_id, name, dept_name, dept_location)  -- Bad

-- Use separate tables with foreign keys
-- departments (dept_id, dept_name, location)          -- Good
-- employees (emp_id, name, dept_id)                    -- Good
```

## 2. Query Optimization Tips

```
sql
```

```sql
-- Use EXPLAIN to analyze queries
EXPLAIN FORMAT=JSON
SELECT e.first_name, e.last_name, d.dept_name
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE e.salary > 70000;

-- Avoid SELECT * in production
-- Bad
SELECT * FROM employees WHERE dept_id = 2;

-- Good
SELECT emp_id, first_name, last_name, salary FROM employees WHERE dept_id = 2;

-- Use LIMIT for large result sets
SELECT first_name, last_name, hire_date
FROM employees
ORDER BY hire_date DESC
LIMIT 10;

-- Use EXISTS instead of IN for large subqueries
-- Less efficient
SELECT * FROM employees
WHERE dept_id IN (SELECT dept_id FROM departments WHERE location = 'New York');

-- More efficient
SELECT * FROM employees e
WHERE EXISTS (
    SELECT 1 FROM departments d
    WHERE d.dept_id = e.dept_id AND d.location = 'New York'
);
```

## 3. Security Best Practices

sql

```sql
-- Use parameterized queries (in application code)
-- Instead of: SELECT * FROM employees WHERE emp_id = '" + user_input + "'"
-- Use prepared statements with parameters

-- Principle of least privilege
CREATE USER 'app_user'@'localhost' IDENTIFIED BY 'strong_password';
GRANT SELECT, INSERT, UPDATE ON employee_management.employees TO 'app_user'@'localhost';
-- Don't grant unnecessary privileges

-- Regular security maintenance
-- Update passwords regularly
ALTER USER 'app_user'@'localhost' IDENTIFIED BY 'new_strong_password';

-- Remove unused users
DROP USER 'old_user'@'localhost';

-- Monitor failed login attempts
-- Enable general log to track access
-- SET GLOBAL general_log = 'ON';
```

## 4. Maintenance Tasks

```sql
-- Regular maintenance procedures
-- Schedule these tasks during low-usage periods

-- Update table statistics
ANALYZE TABLE employees, departments, projects;

-- Defragment tables
OPTIMIZE TABLE employees;

-- Check for corruption
CHECK TABLE employees EXTENDED;

-- Update configuration for better performance
-- In my.cnf:
-- innodb_buffer_pool_size = 70% of available RAM
-- query_cache_size = 256M (if using MySQL < 8.0)
-- max_connections = appropriate for your application
```

---

## Common Interview Questions and Answers

### 1. Difference between DDL, DML, DQL, and DCL

**DDL (Data Definition Language):**

- Commands: CREATE, ALTER, DROP, TRUNCATE
- Purpose: Define and modify database structure
- Example: `CREATE TABLE employees (...)`

**DML (Data Manipulation Language):**

- Commands: INSERT, UPDATE, DELETE
- Purpose: Manipulate data within tables
- Example: `INSERT INTO employees VALUES (...)`

**DQL (Data Query Language):**

- Commands: SELECT
- Purpose: Retrieve data from database
- Example: `SELECT * FROM employees WHERE salary > 50000`

**DCL (Data Control Language):**

- Commands: GRANT, REVOKE
- Purpose: Control access to database
- Example: `GRANT SELECT ON employees TO 'user'@'localhost'`

## 2. ACID Properties Example

```sql
-- Atomicity: All operations in transaction succeed or all fail
START TRANSACTION;
UPDATE accounts SET balance = balance - 1000 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 1000 WHERE account_id = 2;
COMMIT; -- Both updates succeed or both fail

-- Consistency: Database remains in valid state
-- Foreign key constraints ensure referential integrity
ALTER TABLE employees ADD CONSTRAINT fk_dept
FOREIGN KEY (dept_id) REFERENCES departments(dept_id);

-- Isolation: Transactions don't interfere with each other
-- Different isolation levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALI
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Durability: Committed changes persist
-- InnoDB storage engine ensures durability through write-ahead logging
```

### 3. Indexing Strategy Example

```sql
-- When to create indexes:
-- 1. Primary keys (automatic)
-- 2. Foreign keys
CREATE INDEX idx_emp_dept ON employees(dept_id);

-- 3. Frequently searched columns
CREATE INDEX idx_emp_email ON employees(email);

-- 4. Columns used in ORDER BY
CREATE INDEX idx_emp_hire_date ON employees(hire_date);

-- 5. Composite indexes for multi-column searches
CREATE INDEX idx_emp_dept_salary ON employees(dept_id, salary);

-- When NOT to create indexes:
-- 1. Small tables (< 1000 rows)
-- 2. Columns that change frequently
-- 3. Tables with high INSERT/UPDATE/DELETE activity
```

---

## Conclusion

This comprehensive guide covers all essential MySQL concepts using a practical employee management system example. The key areas covered include:

1. **Database Design**: Proper normalization and relationship modeling
2. **DDL Operations**: Creating and modifying database structures
3. **DML Operations**: Inserting, updating, and deleting data
4. **DQL Mastery**: Complex queries, joins, subqueries, and window functions

5. **Advanced Features**: Views, stored procedures, triggers, and JSON support
6. **Performance**: Indexing strategies and query optimization
7. **Security**: User management and access control
8. **Maintenance**: Backup, recovery, and monitoring

### Next Steps for Students:

1. **Practice**: Set up the employee management database and run all examples
2. **Experiment**: Modify queries and observe the results
3. **Real Projects**: Apply these concepts to your own database projects
4. **Advanced Topics**: Explore MySQL 8.0 features like CTEs and window functions
5. **Performance Tuning**: Learn to use EXPLAIN and optimize slow queries

### Additional Resources:

- MySQL Official Documentation
- MySQL Workbench for visual database design
- Performance monitoring tools like MySQL Enterprise Monitor
- Practice platforms like SQLBolt, W3Schools, and LeetCode SQL problems

Remember: The best way to learn MySQL is through hands-on practice. Start with simple queries and gradually work your way up to complex operations. Good luck with your MySQL journey!