Database Systems
Lab 5: View, Trigger, Store Procedure, Function, Cursor

Student Name

December 6, 2025

Contents

1 Introduction

This lab report covers the implementation of Views, Triggers, Stored Procedures, Functions, and Cursors in MySQL. The exercises are based on the COMPANY database schema and a Hotel reservation system.

1.1 Database Schema: COMPANY

The COMPANY database consists of the following tables:

- `EMPLOYEE` - Employee information
- `DEPARTMENT` - Department information
- `DEPT_LOCATIONS` - Department locations
- `PROJECT` - Project information
- `WORKS_ON` - Employee-Project assignments
- `DEPENDENT` - Employee dependents

2 Views

Exercise: Specify the following views in SQL on the COMPANY database schema:

a. A view that has the department name, manager name, and manager salary for every department.

b. A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department.

c. A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project.

d. A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project with more than two employees working on it.

e. A view (SSN, Full Name of employee, Number of dependents) that includes information about employees who have the number of dependents greater than 2.

f. A view (Full Name of employee, date of birth, gender) for those employees who have their birthdate in July.

g. A view (Name of dependent, SSN of employee, date of birth of dependent) that includes information on all dependents who are less than 18 years old.

2.1 View (a): Department Manager Information

Requirement: A view that has the department name, manager name, and manager salary for every department.

```sql
DROP VIEW IF EXISTS DepartmentManagerInfo;
CREATE VIEW DepartmentManagerInfo AS
SELECT
    d.Dname AS Department_Name,
    CONCAT(e.Fname, ' ', e.Minit, ' ', e.Lname) AS Manager_Name,
    e.Salary AS Manager_Salary
FROM DEPARTMENT d
JOIN EMPLOYEE e ON d.Mgr_ssn = e.Ssn;
```

Explanation: This view joins the DEPARTMENT and EMPLOYEE tables using the manager's SSN to retrieve the department name, manager's full name (concatenated), and the manager's salary.

Test Validation

```sql
-- Query the view
SELECT * FROM DepartmentManagerInfo;
```

Expected Output:

Department_Name	Manager_Name	Manager_Salary
Research	Franklin T Wong	40000.00
Administration	Jennifer S Wallace	43000.00
Headquarters	James E Borg	55000.00

2.2 View (b): Research Department Employees and Supervisors

Requirement: A view that has the employee name, supervisor name, and employee salary for each employee who works in the 'Research' department.

```sql
DROP VIEW IF EXISTS ResearchEmployeeSupervisor;
CREATE VIEW ResearchEmployeeSupervisor AS
SELECT
    CONCAT(e.Fname, ' ', e.Minit, ' ', e.Lname) AS Employee_Name,
    CONCAT(s.Fname, ' ', s.Minit, ' ', s.Lname) AS Supervisor_Name,
    e.Salary AS Employee_Salary
FROM EMPLOYEE e
LEFT JOIN EMPLOYEE s ON e.Super_ssn = s.Ssn
JOIN DEPARTMENT d ON e.Dno = d.Dnumber
WHERE d.Dname = 'Research';
```

Explanation: This view uses a self-join on the EMPLOYEE table to get supervisor information, with a LEFT JOIN to handle employees without supervisors. The WHERE clause filters for the Research department.

Test Validation

```sql
-- Query the view
SELECT * FROM ResearchEmployeeSupervisor;
```

Expected Output:

Employee_Name	Supervisor_Name	Employee_Salary
John B Smith	Franklin T Wong	30000.00
Franklin T Wong	James E Borg	40000.00
Ramesh K Narayan	Franklin T Wong	38000.00
Joyce A English	Franklin T Wong	25000.00
Ahmad V Jabbar	Franklin T Wong	25000.00

2.3 View (c): Project Information

Requirement: A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project.

```sql
DROP VIEW IF EXISTS ProjectInfo;
CREATE VIEW ProjectInfo AS
SELECT
    p.Pname AS Project_Name,
    d.Dname AS Controlling_Department,
    COUNT(w.Essn) AS Number_of_Employees,
    SUM(IFNULL(w.Hours, 0)) AS Total_Hours_Per_Week
FROM PROJECT p
JOIN DEPARTMENT d ON p.Dnum = d.Dnumber
LEFT JOIN WORKS_ON w ON p.Pnumber = w.Pno
GROUP BY p.Pnumber, p.Pname, d.Dname;
```

Explanation: This view joins PROJECT, DEPARTMENT, and WORKS_ON tables, using GROUP BY to aggregate employee counts and total hours per project.

Test Validation

```sql
-- Query the view
SELECT * FROM ProjectInfo;
```

Expected Output:

Project_Name	Ctrl_Dept	Num_Emp	Total_Hours
ProductX	Research	2	52.5
ProductY	Research	3	37.5
ProductZ	Research	2	50.0
Computerization	Administration	3	55.0
Reorganization	Headquarters	3	25.0
Newbenefits	Administration	3	55.0

2.4 View (d): Projects with More Than Two Employees

Requirement: A view that has the project name, controlling department name, number of employees, and total hours worked per week on the project for each project with more than two employees working on it.

```sql
DROP VIEW IF EXISTS ProjectInfoMoreThanTwo;
CREATE VIEW ProjectInfoMoreThanTwo AS
SELECT
    p.Pname AS Project_Name,
    d.Dname AS Controlling_Department,
    COUNT(w.Essn) AS Number_of_Employees,
    SUM(IFNULL(w.Hours, 0)) AS Total_Hours_Per_Week
FROM PROJECT p
JOIN DEPARTMENT d ON p.Dnum = d.Dnumber
LEFT JOIN WORKS_ON w ON p.Pnumber = w.Pno
GROUP BY p.Pnumber, p.Pname, d.Dname
HAVING COUNT(w.Essn) > 2;
```

Explanation: Similar to View (c), but with a HAVING clause to filter projects that have more than 2 employees.

Test Validation

```sql
-- Query the view
SELECT * FROM ProjectInfoMoreThanTwo;
```

Expected Output:

Project_Name	Ctrl_Dept	Num_Emp	Total_Hours
ProductY	Research	3	37.5
Computerization	Administration	3	55.0
Reorganization	Headquarters	3	25.0
Newbenefits	Administration	3	55.0

2.5 View (e): Employees with More Than 2 Dependents

Requirement: A view (SSN, Full Name of employee, Number of dependents) that includes information about employees who have the number of dependents greater than 2.

```sql
DROP VIEW IF EXISTS EmployeesWithManyDependents;
CREATE VIEW EmployeesWithManyDependents AS
SELECT
    e.Ssn AS SSN,
    CONCAT(e.Fname, ' ', e.Minit, ' ', e.Lname) AS Full_Name,
    COUNT(dep.Dependent_name) AS Number_of_Dependents
FROM EMPLOYEE e
JOIN DEPENDENT dep ON e.Ssn = dep.Essn
GROUP BY e.Ssn, e.Fname, e.Minit, e.Lname
HAVING COUNT(dep.Dependent_name) > 2;
```

Explanation: This view joins EMPLOYEE and DEPENDENT tables, groups by employee, and filters those with more than 2 dependents using HAVING.

Test Validation

```sql
-- Query the view
SELECT * FROM EmployeesWithManyDependents;
```

Expected Output:

SSN	Full_Name	Number_of_Dependents
333445555	Franklin T Wong	3
123456789	John B Smith	3

2.6 View (f): July Birthday Employees

Requirement: A view (Full Name of employee, date of birth, gender) for those employees who have their birthdate in July.

```sql
DROP VIEW IF EXISTS JulyBirthdayEmployees;
CREATE VIEW JulyBirthdayEmployees AS
SELECT
    CONCAT(e.Fname, ' ', e.Minit, ' ', e.Lname) AS Full_Name,
    e.Bdate AS Date_of_Birth,
    e.Sex AS Gender
FROM EMPLOYEE e
WHERE MONTH(e.Bdate) = 7;
```

Explanation: This view uses the MONTH() function to filter employees born in July (month 7).

Test Validation

```sql
-- Query the view
SELECT * FROM JulyBirthdayEmployees;
```

Expected Output:

Full_Name	Date_of_Birth	Gender
Joyce A English	1972-07-31	F
Alicia J Zelaya	1968-07-19	F

2.7 View (g): Young Dependents (Under 18)

Requirement: A view (Name of dependent, SSN of employee, date of birth of dependent) that includes information on all dependents who are less than 18 years old.

```sql
DROP VIEW IF EXISTS YoungDependents;
CREATE VIEW YoungDependents AS
SELECT
    dep.Dependent_name AS Dependent_Name,
    dep.Essn AS Employee_SSN,
    dep.Bdate AS Dependent_Date_of_Birth
FROM DEPENDENT dep
WHERE TIMESTAMPDIFF(YEAR, dep.Bdate, CURDATE()) < 18;
```

Explanation: This view uses TIMESTAMPDIFF() to calculate the age of dependents and filters those under 18 years old.

Test Validation

```sql
-- Query the view
SELECT * FROM YoungDependents;
```

Expected Output: Empty set (no dependents under 18 in current data).

3 Trigger (a) - Business Rules

Exercise: Create a database trigger for the following situations:

- The supervisor of an employee must be older than the employee.
- The salary of an employee cannot be greater than the salary of his/her supervisor.
- The salary of an employee can only increase.
- When increasing salary of employee, the increasing amount must not be more than 20% of current salary.
- An employee works on at most 4 projects.
- The maximum number of hours an employee can work on all projects per week is 56.
- The location of a project must be one of the locations of its department.
- The salary of a department manager must be higher than the other employees working for that department.
- Only department managers can work less than 5 hours on a project.

3.1 Trigger (a.1): Supervisor Must Be Older

Requirement: The supervisor of an employee must be older than the employee.

```
DROP TRIGGER IF EXISTS trg_supervisor_older_insert;
DELIMITER //
CREATE TRIGGER trg_supervisor_older_insert
BEFORE INSERT ON EMPLOYEE
FOR EACH ROW
BEGIN
    DECLARE supervisor_bdate DATE;
    IF NEW.Super_ssn IS NOT NULL THEN
        SELECT Bdate INTO supervisor_bdate
        FROM EMPLOYEE WHERE Ssn = NEW.Super_ssn;
        IF supervisor_bdate IS NOT NULL AND NEW.Bdate <= supervisor_bdate THEN
            SIGNAL SQLSTATE ’45000’
            SET MESSAGE_TEXT = ’Error: Supervisor must be older than the employee.’;
        END IF;
    END IF;
END //
DELIMITER ;
```

Explanation: This trigger checks the birthdate of the supervisor before inserting an employee. If the supervisor is not older, it raises an error using SIGNAL.

Test Validation

```sql
-- VALID INSERT: Employee born 1990, Supervisor (333445555) born 1955
INSERT INTO EMPLOYEE VALUES
('Test', 'A', 'Valid', '111111110', '1990-01-01', '123 Test St', 'M', 25000, '333445555', 5);
-- Result: Success

-- INVALID INSERT: Employee born 1940, Supervisor born 1955 (supervisor younger)
INSERT INTO EMPLOYEE VALUES
('Test', 'B', 'Invalid', '111111111', '1940-01-01', '123 Test St', 'M', 25000, '333445555', 5);
-- Result: Error - Supervisor must be older than the employee.
```

3.2 Trigger (a.2): Salary Cannot Exceed Supervisor's Salary

Requirement: The salary of an employee cannot be greater than the salary of his/her supervisor.

```sql
DROP TRIGGER IF EXISTS trg_salary_less_than_supervisor_insert;
DELIMITER //
CREATE TRIGGER trg_salary_less_than_supervisor_insert
BEFORE INSERT ON EMPLOYEE
FOR EACH ROW
BEGIN
    DECLARE supervisor_salary DECIMAL(10, 2);
    IF NEW.Super_ssn IS NOT NULL THEN
        SELECT Salary INTO supervisor_salary
        FROM EMPLOYEE WHERE Ssn = NEW.Super_ssn;
        IF supervisor_salary IS NOT NULL AND NEW.Salary > supervisor_salary THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Error: Employee salary cannot be greater than supervisor salary.';
        END IF;
    END IF;
END //
DELIMITER ;
```

Test Validation

```sql
-- VALID INSERT: Employee salary 35000, Supervisor salary 40000
INSERT INTO EMPLOYEE VALUES
('Test', 'C', 'Valid', '111111112', '1990-01-01', '123 Test St', 'M', 35000, '333445555', 5);
-- Result: Success

-- INVALID INSERT: Employee salary 50000 > Supervisor salary 40000
INSERT INTO EMPLOYEE VALUES
('Test', 'D', 'Invalid', '111111113', '1990-01-01', '123 Test St', 'M', 50000, '333445555', 5);
-- Result: Error - Employee salary cannot be greater than supervisor salary.
```

3.3 Trigger (a.3): Salary Can Only Increase

Requirement: The salary of an employee can only increase.

```sql
DROP TRIGGER IF EXISTS trg_salary_only_increase;
DELIMITER //
CREATE TRIGGER trg_salary_only_increase
BEFORE UPDATE ON EMPLOYEE
FOR EACH ROW
BEGIN
    IF NEW.Salary < OLD.Salary THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Employee salary can only increase, not decrease.';
    END IF;
END //
DELIMITER ;
```

Test Validation

```sql
-- VALID UPDATE: Increase salary from 30000 to 32000
UPDATE EMPLOYEE SET Salary = 32000 WHERE Ssn = '123456789';
-- Result: Success

-- INVALID UPDATE: Decrease salary from 32000 to 28000
UPDATE EMPLOYEE SET Salary = 28000 WHERE Ssn = '123456789';
-- Result: Error - Employee salary can only increase, not decrease.
```

3.4 Trigger (a.4): Maximum 20% Salary Increase

Requirement: When increasing salary of employee, the increasing amount must not be more than 20% of current salary.

```sql
DROP TRIGGER IF EXISTS trg_salary_increase_max_20_percent;
DELIMITER //
CREATE TRIGGER trg_salary_increase_max_20_percent
BEFORE UPDATE ON EMPLOYEE
FOR EACH ROW
BEGIN
    IF NEW.Salary > OLD.Salary THEN
        IF (NEW.Salary - OLD.Salary) > (OLD.Salary * 0.20) THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Error: Salary increase cannot exceed 20% of current salary.';
        END IF;
    END IF;
END //
DELIMITER ;
```

Test Validation

```sql
-- VALID UPDATE: Increase salary by 15% (30000 -> 34500)
UPDATE EMPLOYEE SET Salary = 34500 WHERE Ssn = '123456789';
-- Result: Success

-- INVALID UPDATE: Increase salary by 50% (30000 -> 45000)
UPDATE EMPLOYEE SET Salary = 45000 WHERE Ssn = '123456789';
-- Result: Error - Salary increase cannot exceed 20% of current salary.
```

3.5 Trigger (a.5): Maximum 4 Projects Per Employee

Requirement: An employee works on at most 4 projects.

```
DROP TRIGGER IF EXISTS trg_max_4_projects_insert;
DELIMITER //
CREATE TRIGGER trg_max_4_projects_insert
BEFORE INSERT ON WORKS_ON
FOR EACH ROW
BEGIN
    DECLARE project_count INT;
    SELECT COUNT(*) INTO project_count FROM WORKS_ON WHERE Essn = NEW.
   Essn;
    IF project_count >= 4 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: An employee can work on at most 4
   projects.';
    END IF;
END //
DELIMITER ;
```

Test Validation

```
-- Check current projects for employee '123456789'
SELECT Essn, Pno FROM WORKS_ON WHERE Essn = '123456789';

-- INVALID INSERT: Adding 5th project (employee already has 4 projects)
-- Note: Use existing project number (10, 20, 30 exist)
INSERT INTO WORKS_ON VALUES ('123456789', 10, 10);
-- Result: Error - An employee can work on at most 4 projects.

-- VALID INSERT: Employee with less than 4 projects (666884444 has
   fewer projects)
-- Check available projects for employee in dept 5
SELECT Pnumber FROM PROJECT WHERE Pnumber NOT IN (SELECT Pno FROM
   WORKS_ON WHERE Essn = '666884444') AND Dnum = 5;
INSERT INTO WORKS_ON VALUES ('666884444', 1, 10);
-- Result: Success
```

3.6 Trigger (a.6): Maximum 56 Hours Per Week

Requirement: The maximum number of hours an employee can work on all projects per week is 56.

```
DROP TRIGGER IF EXISTS trg_max_56_hours_insert;
DELIMITER //
CREATE TRIGGER trg_max_56_hours_insert
BEFORE INSERT ON WORKS_ON
FOR EACH ROW
BEGIN
    DECLARE total_hours DECIMAL(5, 1);
```

```sql
    SELECT IFNULL(SUM(Hours), 0) INTO total_hours
    FROM WORKS_ON WHERE Essn = NEW.Essn;
    IF (total_hours + IFNULL(NEW.Hours, 0)) > 56 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Total hours per week cannot exceed 56.';
    END IF;
END //
DELIMITER ;
```

Test Validation

```sql
-- Check current hours for employee '123456789'
SELECT Essn, SUM(Hours) AS Total_Hours FROM WORKS_ON WHERE Essn = '123456789';

-- VALID INSERT: Adding hours that don't exceed 56 total
-- Note: Use existing project number that employee isn't working on yet (project 3, dept 5)
INSERT INTO WORKS_ON VALUES ('123456789', 3, 5);
-- Result: Success

-- INVALID INSERT: Adding hours that would exceed 56
-- First check if this would exceed: existing hours + new hours > 56
INSERT INTO WORKS_ON VALUES ('666884444', 2, 50);
-- Result: Error - Total hours per week cannot exceed 56.
```

3.7 Trigger (a.7): Project Location Must Match Department Location

Requirement: The location of a project must be one of the locations of its department.

```sql
-- Trigger 1: Ensure employees only work on projects from their department
DROP TRIGGER IF EXISTS trg_project_dept_valid;
DELIMITER //
CREATE TRIGGER trg_project_dept_valid
BEFORE INSERT ON WORKS_ON
FOR EACH ROW
BEGIN
    DECLARE emp_dept INT;
    DECLARE project_dept INT;

    SELECT Dno INTO emp_dept FROM EMPLOYEE WHERE Ssn = NEW.Essn;
    SELECT Dnum INTO project_dept FROM PROJECT WHERE Pnumber = NEW.Pno;

    IF emp_dept != project_dept THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Employee can only work on projects controlled by their department.';
    END IF;
END //
DELIMITER ;

-- Trigger 2: Ensure project location matches department location
DROP TRIGGER IF EXISTS trg_project_location_valid;
```

```sql
DELIMITER //
CREATE TRIGGER trg_project_location_valid
BEFORE INSERT ON PROJECT
FOR EACH ROW
BEGIN
    DECLARE location_exists INT;

    SELECT COUNT(*) INTO location_exists
    FROM DEPT_LOCATIONS
    WHERE Dnumber = NEW.Dnum AND Dlocation = NEW.Plocation;

    IF location_exists = 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Error: Project location must be one of its department locations.';
    END IF;
END //
DELIMITER ;
```

Explanation: Two triggers enforce this constraint: the first ensures employees work on projects from their department, and the second validates that when inserting a new project, its location is one of the department's registered locations.

Test Validation

```sql
-- Check Dept 5 locations
SELECT Dnumber, Dlocation FROM DEPT_LOCATIONS WHERE Dnumber = 5;

-- VALID INSERT: Project in valid department location
INSERT INTO PROJECT VALUES ('TestProject', 99, 'Houston', 5);
-- Result: Success

-- INVALID INSERT: Project in location not belonging to department
INSERT INTO PROJECT VALUES ('BadProject', 100, 'New York', 5);
-- Result: Error - Project location must be one of its department locations.
```

3.8 Trigger (a.8): Manager Salary Must Be Highest

Requirement: The salary of a department manager must be higher than the other employees working for that department.

```sql
DROP TRIGGER IF EXISTS trg_manager_salary_highest_insert;
DELIMITER //
CREATE TRIGGER trg_manager_salary_highest_insert
BEFORE INSERT ON EMPLOYEE
FOR EACH ROW
BEGIN
    DECLARE manager_salary DECIMAL(10, 2);
    DECLARE manager_ssn CHAR(9);

    IF NEW.Dno IS NOT NULL THEN
        SELECT Mgr_ssn INTO manager_ssn FROM DEPARTMENT WHERE Dnumber = NEW.Dno;
        IF manager_ssn IS NOT NULL AND manager_ssn != NEW.Ssn THEN
            SELECT Salary INTO manager_salary FROM EMPLOYEE WHERE Ssn = manager_ssn;
            IF manager_salary IS NOT NULL AND NEW.Salary >= manager_salary THEN
                SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'Error: Employee salary cannot be equal or greater than department manager salary.';
            END IF;
        END IF;
    END IF;
END //
DELIMITER ;
```

Test Validation

```sql
-- Check Dept 5 manager and salaries
SELECT Mgr_ssn, (SELECT Salary FROM EMPLOYEE WHERE Ssn = Mgr_ssn) AS Manager_Salary
FROM DEPARTMENT WHERE Dnumber = 5;

-- VALID INSERT: Employee salary < Manager salary
INSERT INTO EMPLOYEE VALUES
('Test', 'E', 'Valid', '111111114', '1990-01-01', '123 St', 'M', 35000, '333445555', 5);
-- Result: Success

-- INVALID INSERT: Employee salary >= Manager salary
INSERT INTO EMPLOYEE VALUES
('Test', 'F', 'Invalid', '111111115', '1990-01-01', '123 St', 'M', 50000, '333445555', 5);
-- Result: Error - Employee salary cannot be equal or greater than manager salary.
```

3.9 Trigger (a.9): Only Managers Can Work Less Than 5 Hours

Requirement: Only department managers can work less than 5 hours on a project.

```sql
DROP TRIGGER IF EXISTS trg_min_5_hours_non_manager_insert;
DELIMITER //
CREATE TRIGGER trg_min_5_hours_non_manager_insert
BEFORE INSERT ON WORKS_ON
FOR EACH ROW
BEGIN
    DECLARE is_manager INT;
    IF NEW.Hours IS NOT NULL AND NEW.Hours < 5 THEN
        SELECT COUNT(*) INTO is_manager FROM DEPARTMENT WHERE Mgr_ssn = NEW.Essn;
        IF is_manager = 0 THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Error: Only department managers can work less than 5 hours on a project.';
        END IF;
    END IF;
END //
DELIMITER ;
```

Test Validation

```
-- Check which projects each person works on
SELECT Essn, COUNT(*) AS Project_Count FROM WORKS_ON GROUP BY Essn;

-- VALID INSERT: Manager (888665555, Dept 1 manager) working 3 hours on
    project 20 (dept 1)
-- Manager 888665555 currently works only on project 20 with NULL hours
UPDATE WORKS_ON SET Hours = 3 WHERE Essn = '888665555' AND Pno = 20;
-- Result: Success (managers can work < 5 hours)

-- INVALID INSERT: Non-manager (453453453) working 3 hours on project 3
    from dept 5
INSERT INTO WORKS_ON VALUES ('453453453', 3, 3);
-- Result: Error - Only department managers can work less than 5 hours.

-- VALID INSERT: Non-manager working 10 hours
INSERT INTO WORKS_ON VALUES ('453453453', 3, 10);
-- Result: Success
```

4 Task (b) - Num_of_Emp Derived Attribute

Exercise: Alter table Department to add the attribute Num_of_Emp that stores the number of employees working for each department. This attribute is a derived attribute from Employee.DNO and its value must be automatically calculated.

Solution:

```sql
-- Add the column
ALTER TABLE DEPARTMENT ADD COLUMN Num_of_Emp INT DEFAULT 0;

-- Initialize the column with current counts
UPDATE DEPARTMENT d
SET Num_of_Emp = (SELECT COUNT(*) FROM EMPLOYEE e WHERE e.Dno = d.Dnumber);

-- Trigger for INSERT
DROP TRIGGER IF EXISTS trg_update_num_emp_insert;
DELIMITER //
CREATE TRIGGER trg_update_num_emp_insert
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
BEGIN
    IF NEW.Dno IS NOT NULL THEN
        UPDATE DEPARTMENT
        SET Num_of_Emp = Num_of_Emp + 1
        WHERE Dnumber = NEW.Dno;
    END IF;
END //
DELIMITER ;

-- Trigger for DELETE
DROP TRIGGER IF EXISTS trg_update_num_emp_delete;
DELIMITER //
CREATE TRIGGER trg_update_num_emp_delete
AFTER DELETE ON EMPLOYEE
FOR EACH ROW
BEGIN
    IF OLD.Dno IS NOT NULL THEN
        UPDATE DEPARTMENT
        SET Num_of_Emp = Num_of_Emp - 1
        WHERE Dnumber = OLD.Dno;
    END IF;
END //
DELIMITER ;

-- Trigger for UPDATE
DROP TRIGGER IF EXISTS trg_update_num_emp_update;
DELIMITER //
CREATE TRIGGER trg_update_num_emp_update
AFTER UPDATE ON EMPLOYEE
FOR EACH ROW
BEGIN
    IF OLD.Dno IS NOT NULL AND (NEW.Dno IS NULL OR OLD.Dno != NEW.Dno) THEN
        UPDATE DEPARTMENT SET Num_of_Emp = Num_of_Emp - 1 WHERE Dnumber = OLD.Dno;
    END IF;
    IF NEW.Dno IS NOT NULL AND (OLD.Dno IS NULL OR OLD.Dno != NEW.Dno)
```

```
   THEN
        UPDATE DEPARTMENT SET Num_of_Emp = Num_of_Emp + 1 WHERE Dnumber
    = NEW.Dno;
    END IF;
END //
DELIMITER ;
```

Test Validation

```
-- Check current department counts
SELECT Dnumber, Dname, Num_of_Emp FROM DEPARTMENT;
```

Expected Output:

Dnumber	Dname	Num_of_Emp
1	Headquarters	1
4	Administration	2
5	Research	5

```
-- Test INSERT: Add new employee to Dept 5
INSERT INTO EMPLOYEE VALUES (’New’, ’N’, ’Emp’, ’999999999’, ’
   1990-01-01’,
                             ’123 St’, ’M’, 25000, ’333445555’, 5);
SELECT Dnumber, Num_of_Emp FROM DEPARTMENT WHERE Dnumber = 5;
-- Result: Num_of_Emp = 6 (incremented from 5)

-- Test DELETE: Remove the employee
DELETE FROM EMPLOYEE WHERE Ssn = ’999999999’;
SELECT Dnumber, Num_of_Emp FROM DEPARTMENT WHERE Dnumber = 5;
-- Result: Num_of_Emp = 5 (decremented back)
```

5 Function (c) - Get Total Projects

Exercise: Write a function that returns the total number of projects when given an employee's ID.

- **Input:** employee ID
- **Output:** total number of projects

Solution:

```sql
DROP FUNCTION IF EXISTS GetTotalProjectsForEmployee;
DELIMITER //
CREATE FUNCTION GetTotalProjectsForEmployee(emp_ssn CHAR(9))
RETURNS INT
DETERMINISTIC
READS SQL DATA
BEGIN
    DECLARE total_projects INT;

    SELECT COUNT(*) INTO total_projects
    FROM WORKS_ON
    WHERE Essn = emp_ssn;

    RETURN total_projects;
END //
DELIMITER ;

-- Example usage:
SELECT GetTotalProjectsForEmployee('123456789') AS Total_Projects;
SELECT GetTotalProjectsForEmployee('333445555') AS Total_Projects;
```

Explanation: This function takes an employee SSN as input and returns the count of projects that employee works on from the WORKS_ON table.

Test Validation

```sql
-- Call function for employee '123456789'
SELECT GetTotalProjectsForEmployee('123456789') AS Total_Projects;
```

Expected Output:

Total_Projects
2

```sql
-- Call function for employee '333445555'
SELECT GetTotalProjectsForEmployee('333445555') AS Total_Projects;
```

Expected Output:

Total_Projects
2

```sql
-- List all employees with their project counts
SELECT Ssn, CONCAT(Fname, ' ', Lname) AS Name,
       GetTotalProjectsForEmployee(Ssn) AS Projects
FROM EMPLOYEE ORDER BY Projects DESC;
```

Expected Output:

Ssn	**Name**	**Projects**
123456789	John Smith	2
333445555	Franklin Wong	2
999887777	Alicia Zelaya	2
987654321	Jennifer Wallace	2
666884444	Ramesh Narayan	3
453453453	Joyce English	2
987987987	Ahmad Jabbar	2
888665555	James Borg	0

6 Procedure (d) - Print Employee Details

Exercise: Create a stored procedure that prints SSN, Full name, Department name, and annual salary of all employees.

Solution:

```sql
DROP PROCEDURE IF EXISTS PrintEmployeeDetails;
DELIMITER //
CREATE PROCEDURE PrintEmployeeDetails()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE v_ssn CHAR(9);
    DECLARE v_fullname VARCHAR(50);
    DECLARE v_dname VARCHAR(25);
    DECLARE v_annual_salary DECIMAL(12, 2);

    DECLARE emp_cursor CURSOR FOR
        SELECT
            e.Ssn,
            CONCAT(e.Fname, ' ', e.Minit, ' ', e.Lname) AS Full_Name,
            d.Dname,
            e.Salary * 12 AS Annual_Salary
        FROM EMPLOYEE e
        LEFT JOIN DEPARTMENT d ON e.Dno = d.Dnumber;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    DROP TEMPORARY TABLE IF EXISTS temp_employee_details;
    CREATE TEMPORARY TABLE temp_employee_details (
        SSN CHAR(9),
        Full_Name VARCHAR(50),
        Department_Name VARCHAR(25),
        Annual_Salary DECIMAL(12, 2)
    );

    OPEN emp_cursor;

    read_loop: LOOP
        FETCH emp_cursor INTO v_ssn, v_fullname, v_dname, v_annual_salary;
        IF done THEN
            LEAVE read_loop;
        END IF;
        INSERT INTO temp_employee_details
        VALUES (v_ssn, v_fullname, v_dname, v_annual_salary);
    END LOOP;

    CLOSE emp_cursor;
    SELECT * FROM temp_employee_details;
    DROP TEMPORARY TABLE IF EXISTS temp_employee_details;
END //
DELIMITER ;
```

Test Validation

```sql
-- Call the procedure
CALL PrintEmployeeDetails();
```

Expected Output:

SSN	Full_Name	Dept_Name	Annual_Salary
123456789	John B Smith	Research	360000.00
333445555	Franklin T Wong	Research	480000.00
999887777	Alicia J Zelaya	Administration	300000.00
987654321	Jennifer S Wallace	Administration	516000.00
666884444	Ramesh K Narayan	Research	456000.00
453453453	Joyce A English	Research	300000.00
987987987	Ahmad V Jabbar	Research	300000.00
888665555	James E Borg	Headquarters	660000.00

7 Trigger (e) - Salary Log

Exercise: Write a trigger that logs any changes in case the new salary is greater than 50000 updated or inserted into our database.

Hint:

- Create a LOG table (SSN, CONTENT, DATE)
- E.g., ('123456789', 'SALARY UPDATE FROM 30000 TO 70000', '06-NOV-2021')

Solution:

```sql
-- Create the LOG table
DROP TABLE IF EXISTS SALARY_LOG;
CREATE TABLE SALARY_LOG (
    Log_id INT AUTO_INCREMENT PRIMARY KEY,
    SSN CHAR(9),
    Content VARCHAR(255),
    Log_Date DATE
);

-- Trigger for INSERT
DROP TRIGGER IF EXISTS trg_log_salary_insert;
DELIMITER //
CREATE TRIGGER trg_log_salary_insert
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
BEGIN
    IF NEW.Salary > 50000 THEN
        INSERT INTO SALARY_LOG (SSN, Content, Log_Date)
        VALUES (NEW.Ssn, CONCAT('SALARY INSERT: ', NEW.Salary), CURDATE());
    END IF;
END //
DELIMITER ;

-- Trigger for UPDATE
DROP TRIGGER IF EXISTS trg_log_salary_update;
DELIMITER //
CREATE TRIGGER trg_log_salary_update
AFTER UPDATE ON EMPLOYEE
FOR EACH ROW
BEGIN
    IF NEW.Salary > 50000 THEN
        INSERT INTO SALARY_LOG (SSN, Content, Log_Date)
        VALUES (NEW.Ssn, CONCAT('SALARY UPDATE FROM ', OLD.Salary, ' TO ', NEW.Salary), CURDATE());
    END IF;
END //
DELIMITER ;
```

Test Validation

```sql
-- Test INSERT with high salary (> 50000)
INSERT INTO EMPLOYEE VALUES ('High', 'H', 'Earner', '888888888', '1980-01-01',
                             '123 St', 'M', 55000, NULL, 1);
```

```sql

-- Check the log
SELECT * FROM SALARY_LOG;
```

Expected Output:

Log_id	SSN	Content	Log_Date
1	888888888	SALARY INSERT: 55000.00	2024-12-05

```sql
-- Test UPDATE with high salary
UPDATE EMPLOYEE SET Salary = 60000 WHERE Ssn = '888888888';
SELECT * FROM SALARY_LOG ORDER BY Log_id DESC LIMIT 1;
```

Expected Output:

Log_id	SSN	Content	Log_Date
2	888888888	SALARY UPDATE: 55000 TO 60000	2024-12-05

8 Procedure (f) - Employee Salary Levels

Exercise: Write a stored procedure that prints out the level of salary for each employee.

Rules:

- if (salary $<$ 20000) then “level C”
- if (salary between 20000 and 50000) then “level B”
- if (salary $>$ 50000) then “level A”

Example Output:

```
123456789, John B Smith, level B
333445555, Franklin T Wong, level B
...
```

Solution:

```sql
DROP PROCEDURE IF EXISTS PrintEmployeeSalaryLevel;
DELIMITER //
CREATE PROCEDURE PrintEmployeeSalaryLevel()
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE v_ssn CHAR(9);
    DECLARE v_fullname VARCHAR(50);
    DECLARE v_salary DECIMAL(10, 2);
    DECLARE v_level VARCHAR(10);

    DECLARE emp_cursor CURSOR FOR
        SELECT
            e.Ssn,
            CONCAT(e.Fname, ' ', e.Minit, ' ', e.Lname) AS Full_Name,
            e.Salary
        FROM EMPLOYEE e;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    DROP TEMPORARY TABLE IF EXISTS temp_salary_levels;
    CREATE TEMPORARY TABLE temp_salary_levels (
        SSN CHAR(9),
        Full_Name VARCHAR(50),
        Salary_Level VARCHAR(10)
    );

    OPEN emp_cursor;

    read_loop: LOOP
        FETCH emp_cursor INTO v_ssn, v_fullname, v_salary;
        IF done THEN
            LEAVE read_loop;
        END IF;

        IF v_salary < 20000 THEN
            SET v_level = 'level C';
        ELSEIF v_salary >= 20000 AND v_salary <= 50000 THEN
            SET v_level = 'level B';
        ELSE
```

```sql
            SET v_level = 'level A';
        END IF;

        INSERT INTO temp_salary_levels VALUES (v_ssn, v_fullname, v_level);
    END LOOP;

    CLOSE emp_cursor;
    SELECT * FROM temp_salary_levels;
    DROP TEMPORARY TABLE IF EXISTS temp_salary_levels;
END //
DELIMITER ;
```

Test Validation

```sql
-- Call the procedure
CALL PrintEmployeeSalaryLevel();
```

Expected Output:

SSN	Full_Name	Salary_Level
123456789	John B Smith	level B
333445555	Franklin T Wong	level B
999887777	Alicia J Zelaya	level B
987654321	Jennifer S Wallace	level B
666884444	Ramesh K Narayan	level B
453453453	Joyce A English	level B
987987987	Ahmad V Jabbar	level B
888665555	James E Borg	level A

Note: The salary levels are determined by:

- level C: salary < 20000
- level B: $20000 \leq$ salary ≤ 50000
- level A: salary > 50000

9 Exercise 2 - Hotel Database Constraints

Exercise: Consider the following schema:

Hotel(hotelNo, hotelName, city)
Room(roomNo, hotelNo, type, price) FK: hotelNo refs Hotel
Booking(hotelNo, guestNo, dateFrom, dateTo, roomNo) FK: hotelNo refs Hotel, guestNo refs Guest, (hotelNo, roomNo) refs Room
Guest(guestNo, guestName, guestAddress)

Write a constraint to implement each of the following requirements. For each constraint you should determine the most appropriate technique from the following:

1) a domain type, domain constraint, attribute constraint
2) a general (table level) constraint
3) an assertion
4) a trigger
5) embedded SQL

9.1 Constraint (a) - Room Price

Requirement: The price of a room must be between $10 and $100.

Solution:

```sql
-- Domain Type / Attribute Constraint
-- This is best implemented using an attribute/column constraint

ALTER TABLE Room
ADD CONSTRAINT chk_room_price
CHECK (price >= 10 AND price <= 100);
```

Test Validation

```sql
-- Test 1: Try to insert room with price < 10 (should fail)
INSERT INTO Room VALUES (301, 1, ’Single’, 5);

-- Test 2: Try to insert room with price > 100 (should fail)
INSERT INTO Room VALUES (302, 1, ’Suite’, 150);

-- Test 3: Insert room with valid price (should succeed)
INSERT INTO Room VALUES (303, 1, ’Double’, 50);
```

Expected Results:

Test	**Action**	**Result**
Test 1	INSERT price=5	Check constraint violated
Test 2	INSERT price=150	Check constraint violated
Test 3	INSERT price=50	Row inserted

9.2 Constraint (b) - Booking Validity

Requirement: A booking cannot be for a period of more than 14 days.

Solution:

```sql
-- Table Level Constraint
-- A CHECK constraint that compares two columns

ALTER TABLE Booking
ADD CONSTRAINT chk_booking_duration
CHECK (DATEDIFF(dateTo, dateFrom) <= 14);
```

Test Validation

```sql
-- Test 1: Try to insert booking for 15 days (should fail)
INSERT INTO Booking VALUES
    (1, 1, '2024-01-01', '2024-01-16', 101);

-- Test 2: Insert booking for exactly 14 days (should succeed)
INSERT INTO Booking VALUES
    (1, 2, '2024-02-01', '2024-02-15', 102);

-- Test 3: Insert booking for 7 days (should succeed)
INSERT INTO Booking VALUES
    (1, 3, '2024-03-01', '2024-03-08', 103);
```

Expected Results:

Test	**Action**	**Result**
Test 1	INSERT 15-day booking	Check constraint violated
Test 2	INSERT 14-day booking	Row inserted
Test 3	INSERT 7-day booking	Row inserted

9.3 Constraint (c) - No Double Booking

Requirement: No room can be double-booked (i.e., the same room in the same hotel cannot be booked by two different guests for the same dates, or dates that would overlap with an existing booking).

Solution:

```sql
-- Trigger (required for complex logic with overlapping dates)
DELIMITER //
CREATE TRIGGER trg_no_double_booking
BEFORE INSERT ON Booking
FOR EACH ROW
BEGIN
    DECLARE conflict_count INT;

    SELECT COUNT(*) INTO conflict_count
    FROM Booking
    WHERE hotelNo = NEW.hotelNo
      AND roomNo = NEW.roomNo
      AND (
          (NEW.dateFrom BETWEEN dateFrom AND dateTo)
          OR (NEW.dateTo BETWEEN dateFrom AND dateTo)
          OR (dateFrom BETWEEN NEW.dateFrom AND NEW.dateTo)
          OR (dateTo BETWEEN NEW.dateFrom AND NEW.dateTo)
      );

    IF conflict_count > 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Room is already booked for overlapping dates';
    END IF;
END //
DELIMITER ;
```

Test Validation

```
-- Setup: Insert initial booking
INSERT INTO Booking VALUES
    (1, 1, '2024-04-10', '2024-04-15', 101);

-- Test 1: Try overlapping booking (should fail)
INSERT INTO Booking VALUES
    (1, 2, '2024-04-12', '2024-04-18', 101);

-- Test 2: Non-overlapping booking (should succeed)
INSERT INTO Booking VALUES
    (1, 2, '2024-04-20', '2024-04-25', 101);
```

Expected Results:

Test	Action	Result
Setup	INSERT Apr 10-15, Room 101	Row inserted
Test 1	INSERT Apr 12-18, Room 101	Trigger error - overlap
Test 2	INSERT Apr 20-25, Room 101	Row inserted

9.4 Constraint (d) - Maximum Grosvenor Bookings

Requirement: No guest can make more than 10 bookings for the same hotel with name ‘Grosvenor’.

Solution:

```
-- Trigger (required for counting bookings per guest per hotel)
DELIMITER //
CREATE TRIGGER trg_max_grosvenor_bookings
BEFORE INSERT ON Booking
FOR EACH ROW
BEGIN
    DECLARE booking_count INT;
    DECLARE hotel_name VARCHAR(50);

    SELECT hotelName INTO hotel_name
    FROM Hotel
    WHERE hotelNo = NEW.hotelNo;

    IF hotel_name = 'Grosvenor' THEN
        SELECT COUNT(*) INTO booking_count
        FROM Booking
        WHERE guestNo = NEW.guestNo
          AND hotelNo = NEW.hotelNo;

        IF booking_count >= 10 THEN
```

```
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Guest cannot make more than 10 bookings
    for Grosvenor hotel';
        END IF;
    END IF;
END //
DELIMITER ;
```

Test Validation

```
-- Setup: Insert Grosvenor hotel
INSERT INTO Hotel VALUES (10, 'Grosvenor', 'London');

-- Insert 10 bookings for guest 1 at Grosvenor
-- (simplified - in practice, dates would vary)
-- After 10 bookings exist...

-- Test: Try to insert 11th booking (should fail)
INSERT INTO Booking VALUES
    (10, 1, '2024-12-01', '2024-12-05', 101);
```

Expected Result: After 10 existing bookings, the 11th booking attempt will fail with the error message “Guest cannot make more than 10 bookings for Grosvenor hotel”.

9.5 Constraint (e) - London Room Increase

Requirement: The price of rooms at hotels in London cannot be increased by more than 10%.

Solution:

```
-- Trigger (required for comparing old and new values on UPDATE)
DELIMITER //
CREATE TRIGGER trg_london_price_increase
BEFORE UPDATE ON Room
FOR EACH ROW
BEGIN
    DECLARE hotel_city VARCHAR(50);

    SELECT city INTO hotel_city
    FROM Hotel
    WHERE hotelNo = NEW.hotelNo;

    IF hotel_city = 'London' AND NEW.price > OLD.price * 1.10 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Price increase for London hotels cannot
   exceed 10%';
    END IF;
END //
DELIMITER ;
```

Test Validation

```
-- Setup: London hotel and room
INSERT INTO Hotel VALUES (20, 'Royal', 'London');
INSERT INTO Room VALUES (201, 20, 'Double', 80);
```

```

-- Test 1: Try to increase price by 15% (should fail)
UPDATE Room SET price = 92 WHERE roomNo = 201 AND hotelNo = 20;

-- Test 2: Increase price by 10% (should succeed)
UPDATE Room SET price = 88 WHERE roomNo = 201 AND hotelNo = 20;
```

Expected Results:

Test	Action	Result
Setup	INSERT Room price=80	Row inserted
Test 1	UPDATE to 92 (15% increase)	Trigger error
Test 2	UPDATE to 88 (10% increase)	Row updated

9.6 Constraint (f) - No Direct Deletes

Requirement: The users cannot directly delete the records from the Hotel table. However, they should still be able to delete records through the Vhotel1 view.

Solution:

This requirement needs a combination of:

1. A trigger to prevent direct DELETEs on the Hotel table
2. An INSTEAD OF trigger on the view to allow deletes through the view

Part 1: Prevent direct deletes on Hotel table

```
-- Trigger to prevent direct deletes on Hotel table
DELIMITER //
CREATE TRIGGER trg_prevent_hotel_delete
BEFORE DELETE ON Hotel
FOR EACH ROW
BEGIN
    -- Check if delete is coming from view (using session variable)
    IF @deleting_from_view IS NULL OR @deleting_from_view = FALSE THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Direct deletion from Hotel table is not
allowed. Use Vhotel1 view instead.';
    END IF;
END //
DELIMITER ;
```

Part 2: Create view and allow deletions through it

```
-- Create the Vhotel1 view
CREATE OR REPLACE VIEW Vhotel1 AS
SELECT hotelNo, hotelName, city FROM Hotel;

-- Stored procedure to delete through view
DELIMITER //
CREATE PROCEDURE DeleteFromVhotel1(IN p_hotelNo INT)
BEGIN
    SET @deleting_from_view = TRUE;
    DELETE FROM Hotel WHERE hotelNo = p_hotelNo;
    SET @deleting_from_view = FALSE;
END //
DELIMITER ;
```

INSTEAD OF Trigger Syntax (Other DBMS)

SQL Server:

```
-- SQL Server supports INSTEAD OF triggers on views
CREATE VIEW Vhotel1 AS
SELECT hotelNo, hotelName, city FROM Hotel;
GO

CREATE TRIGGER trg_vhotel1_delete
ON Vhotel1
INSTEAD OF DELETE
AS
BEGIN
    DELETE FROM Hotel WHERE hotelNo IN (SELECT hotelNo FROM deleted);
END;
GO
```

PostgreSQL:

```
-- PostgreSQL uses trigger functions
CREATE OR REPLACE VIEW Vhotel1 AS
SELECT hotelNo, hotelName, city FROM Hotel;

CREATE OR REPLACE FUNCTION fn_vhotel1_delete()
RETURNS TRIGGER AS $$
BEGIN
    DELETE FROM Hotel WHERE hotelNo = OLD.hotelNo;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_vhotel1_delete
INSTEAD OF DELETE ON Vhotel1
FOR EACH ROW EXECUTE FUNCTION fn_vhotel1_delete();
```

Oracle:

```
-- Oracle supports INSTEAD OF triggers on views
CREATE OR REPLACE VIEW Vhotel1 AS
SELECT hotelNo, hotelName, city FROM Hotel;

CREATE OR REPLACE TRIGGER trg_vhotel1_delete
INSTEAD OF DELETE ON Vhotel1
FOR EACH ROW
BEGIN
    DELETE FROM Hotel WHERE hotelNo = :OLD.hotelNo;
END;
/
```

Test Validation

```
-- Setup: Insert a test hotel
INSERT INTO Hotel VALUES (99, 'Test Hotel', 'Test City');

-- Test 1: Try direct delete (should fail)
DELETE FROM Hotel WHERE hotelNo = 99;

-- Test 2: Delete through view procedure (should succeed)
CALL DeleteFromVhotel1(99);
```

Expected Results:

Test	**Action**	**Result**
Setup	INSERT Hotel 99	Row inserted
Test 1	Direct DELETE	Trigger error
Test 2	CALL DeleteFromVhotel1(99)	Row deleted

10 Conclusion

This laboratory exercise demonstrated the implementation of various database constraints and programming constructs in MySQL. The key concepts covered include:

- **Views:** Created multiple views to simplify complex queries and provide different perspectives on the data, including employee department views, project views, and supervisor/supervisee relationships.
- **Triggers:** Implemented triggers for enforcing business rules (salary limits, supervision rules, project management constraints), maintaining derived attributes, logging changes, and preventing unwanted operations.
- **Stored Functions:** Created user-defined functions to encapsulate reusable logic, such as counting employee project assignments.
- **Stored Procedures:** Developed procedures using cursors and control flow statements to process and display data with formatted output.
- **Constraint Types:** Explored different constraint implementation techniques including:
  - Domain/attribute constraints (CHECK constraints)
  - Table-level constraints
  - Triggers for complex business rules
  - INSTEAD OF triggers for view operations

The exercises also highlighted the differences between various database management systems (MySQL, SQL Server, PostgreSQL, Oracle) in their support for features like INSTEAD OF triggers and assertion constraints.

Understanding these database programming concepts is essential for:

- Maintaining data integrity
- Enforcing business rules at the database level
- Creating efficient and reusable database code
- Designing robust database applications