**1.Consider the following schema for a Library Database:**

**BOOK (Book\_id, Title, Publisher\_Name, Pub\_Year, No\_of\_copies) BOOK\_AUTHORS (Book\_id, Author\_Name)**

**PUBLISHER (Name, Address, Phone)**

**a)Retrieve details of all books in the Book\_id, title, name of publisher, authors**

**b)Get the particulars of borrowers who have borrowed more than 3 books, but from Jan 2017 to Jun2017.**

**c)Delete a book in BOOK table. Update the contents of other tables to reflect this data Manipulation operations.**

**e)Create a view of all books and its number of copies that are currently available in the Library.**

**f)Write a Pl/SQL procedure to display the book details of particular author.**

Schema Setup

CREATE DATABASE library;

USE library;

-- Create Tables

CREATE TABLE PUBLISHER (

Name VARCHAR(100) PRIMARY KEY,

Address VARCHAR(255),

Phone VARCHAR(15)

);

CREATE TABLE BOOK (

Book\_id INT PRIMARY KEY,

Title VARCHAR(255),

Publisher\_Name VARCHAR(100),

Pub\_Year YEAR),

No\_of\_copies INT,

FOREIGN KEY (Publisher\_Name) REFERENCES PUBLISHER(Name)

);

CREATE TABLE BOOK\_AUTHORS (

Book\_id INT,

Author\_Name VARCHAR(100),

PRIMARY KEY (Book\_id, Author\_Name),

FOREIGN KEY (Book\_id) REFERENCES BOOK(Book\_id)

);

-- Additional tables needed for complete functionality

CREATE TABLE BORROWER (

Borrower\_id INT PRIMARY KEY AUTO\_INCREMENT,

Name VARCHAR(100),

Email VARCHAR(100)

);

CREATE TABLE BOOK\_LOAN (

Loan\_id INT PRIMARY KEY AUTO\_INCREMENT,

Book\_id INT,

Borrower\_id INT,

Loan\_date DATE,

Return\_date DATE,

FOREIGN KEY (Book\_id) REFERENCES BOOK(Book\_id),

FOREIGN KEY (Borrower\_id) REFERENCES BORROWER(Borrower\_id)

);

**a) Retrieve Book Details**

SELECT

b.Book\_id,

b.Title,

b.Publisher\_Name,

GROUP\_CONCAT(a.Author\_Name SEPARATOR ', ') AS Authors

FROM BOOK b

JOIN PUBLISHER p ON b.Publisher\_Name = p.Name

LEFT JOIN BOOK\_AUTHORS a ON b.Book\_id = a.Book\_id

GROUP BY b.Book\_id;

**b) Borrowers with >3 Books (Jan-Jun 2017)**

SELECT

br.Borrower\_id,

br.Name,

COUNT(bl.Loan\_id) AS Total\_Borrowed

FROM BORROWER br

JOIN BOOK\_LOAN bl ON br.Borrower\_id = bl.Borrower\_id

WHERE bl.Loan\_date BETWEEN '2017-01-01' AND '2017-06-30'

GROUP BY br.Borrower\_id

HAVING Total\_Borrowed > 3;

**c) Delete a Book and Update Related Tables**

-- Delete from child tables first

DELETE FROM BOOK\_AUTHORS WHERE Book\_id = 101;

DELETE FROM BOOK\_LOAN WHERE Book\_id = 101;

-- Then delete from parent table

DELETE FROM BOOK WHERE Book\_id = 101;

**e) View for Available Books**

CREATE VIEW Available\_Books AS

SELECT

b.Book\_id,

b.Title,

b.No\_of\_copies - COUNT(bl.Loan\_id) AS Available\_Copies

FROM BOOK b

LEFT JOIN BOOK\_LOAN bl ON b.Book\_id = bl.Book\_id

AND bl.Return\_date IS NULL

GROUP BY b.Book\_id;

**f) Stored Procedure for Author's Books**

DELIMITER $$

CREATE PROCEDURE GetBooksByAuthor(IN author\_name VARCHAR(100))

BEGIN

SELECT

b.Book\_id,

b.Title,

b.Publisher\_Name,

b.Pub\_Year,

b.No\_of\_copies

FROM BOOK b

JOIN BOOK\_AUTHORS a ON b.Book\_id = a.Book\_id

WHERE a.Author\_Name = author\_name;

END$$

DELIMITER ;

-- Usage

CALL GetBooksByAuthor('J.K. Rowling');

**2.Create a table employee (S.No,Name,Desination,brach),**

**a)Alter the table by adding a column salary**

**b)Copy the table employee as Emp**

**c)Delete 2nd row from the table**

**d)Drop the table**

**e)Demonstrate the triggers for automatic updation.**

### Step-by-Step Explanation and Code:

**1. Create the**employee**table:**

CREATE TABLE employee (

`S.No` INT PRIMARY KEY,

Name VARCHAR(100),

Designation VARCHAR(100),

Branch VARCHAR(100)

);

**2. Alter the table to add a**salary**column:**

ALTER TABLE employee

ADD COLUMN salary DECIMAL(10, 2);

**3. Copy the**employee**table into a new table**Emp**:**

CREATE TABLE Emp AS SELECT \* FROM employee;

**4. Delete the 2nd row from the table:**

DELETE FROM employee WHERE `S.No` = 2;

**5. Drop the**employee**table:**

DROP TABLE employee;

**6. Demonstrate triggers for automatic updation:**

**Example 1: Trigger to update a**last\_updated**timestamp:**  
First, add the last\_updated column:

ALTER TABLE employee

ADD COLUMN last\_updated TIMESTAMP;

Create a trigger to update last\_updated on row update:

DELIMITER $$

CREATE TRIGGER before\_employee\_update

BEFORE UPDATE ON employee

FOR EACH ROW

BEGIN

SET NEW.last\_updated = CURRENT\_TIMESTAMP();

END$$

DELIMITER ;

**3)Create a table called Employee (Emp\_no Emp\_name, Emp\_dept,Job ,Mgr ,Sal)**

**a)By using the group by clause, display the Emp\_name who belongs to Emp\_dept=”xxx” along with salary**

**b)Display lowest paid employee details under each department**

**c)List the employee names in descending order.**

**d)Rename the column of Employee table using alter command**

**e)Insert row in employee table using Triggers.**

-- 1. Create the Employee table

CREATE TABLE Employee (

Emp\_no INT PRIMARY KEY,

Emp\_name VARCHAR(100),

Emp\_dept VARCHAR(100),

Job VARCHAR(100),

Mgr INT,

Sal DECIMAL(10, 2)

);

-- a) Group By clause (filter by department)

SELECT Emp\_name, Sal

FROM Employee

WHERE Emp\_dept = 'xxx'

GROUP BY Emp\_dept, Emp\_name, Sal;

-- b) Lowest-paid employee per department

SELECT e.\*

FROM Employee e

INNER JOIN (

SELECT Emp\_dept, MIN(Sal) AS min\_sal

FROM Employee

GROUP BY Emp\_dept

) dept\_min

ON e.Emp\_dept = dept\_min.Emp\_dept AND e.Sal = dept\_min.min\_sal;

-- c) Sort names in descending order

SELECT Emp\_name

FROM Employee

ORDER BY Emp\_name DESC;

-- d) Rename column

ALTER TABLE Employee

CHANGE COLUMN Mgr Manager INT;

-- e) Trigger for automatic insertion

CREATE TABLE NewHires (

Emp\_no INT PRIMARY KEY,

Emp\_name VARCHAR(100),

Emp\_dept VARCHAR(100),

Job VARCHAR(100),

Manager INT,

Sal DECIMAL(10, 2)

);

DELIMITER $$

CREATE TRIGGER after\_newhire\_insert

AFTER INSERT ON NewHires

FOR EACH ROW

BEGIN

INSERT INTO Employee (Emp\_no, Emp\_name, Emp\_dept, Job, Manager, Sal)

VALUES (NEW.Emp\_no, NEW.Emp\_name, NEW.Emp\_dept, NEW.Job, NEW.Manager, NEW.Sal);

END$$

DELIMITER ;

**4.Consider the following tables namely “DEPARTMENTS” and “EMPLOYEES”**

**Departments (dept\_no , dept\_ name , dept\_location ),**

**Employees ( emp\_id , emp\_name , emp\_salary,dept\_no).**

**a)Develop a query to grant some privileges of employees table into departments table**

**b)Develop a query to revoke all privileges of employees table from departments table**

**c)Develop a query to revoke some privileges of employees table from departments table**

**d)Write a query to implement the save point.**

**e)Demonstrate the user defined procedure for the above employee database**

### Step-by-Step Explanation and Code:

**1. Create the Tables:**

-- Create DEPARTMENTS table

CREATE TABLE DEPARTMENTS (

dept\_no INT PRIMARY KEY,

dept\_name VARCHAR(100),

dept\_location VARCHAR(100)

);

-- Create EMPLOYEES table with foreign key

CREATE TABLE EMPLOYEES (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(100),

emp\_salary DECIMAL(10, 2),

dept\_no INT,

FOREIGN KEY (dept\_no) REFERENCES DEPARTMENTS(dept\_no)

);

### ****a) Grant Privileges on****EMPLOYEES****Table to a User/Role:****

**Example:** Grant SELECT and INSERT privileges on EMPLOYEES to a user named hr\_manager.  
(Replace *hr\_manager* with your actual username or role.)

GRANT SELECT, INSERT ON EMPLOYEES TO 'hr\_manager'@'localhost';

### ****b) Revoke All Privileges on****EMPLOYEES****Table from a User/Role:****

**Example:** Revoke all privileges from hr\_manager:

REVOKE ALL PRIVILEGES ON EMPLOYEES FROM 'hr\_manager'@'localhost';

### ****c) Revoke Specific Privileges on****EMPLOYEES****Table:****

**Example:** Revoke INSERT privilege from hr\_manager:

REVOKE INSERT ON EMPLOYEES FROM 'hr\_manager'@'localhost';

### ****d) Implement Savepoint in a Transaction:****

START TRANSACTION;

-- Insert a department

INSERT INTO DEPARTMENTS (dept\_no, dept\_name, dept\_location)

VALUES (10, 'IT', 'New York');

-- Create a savepoint

SAVEPOINT savepoint1;

-- Insert an employee

INSERT INTO EMPLOYEES (emp\_id, emp\_name, emp\_salary, dept\_no)

VALUES (101, 'John Doe', 75000, 10);

-- Rollback to savepoint1 (undo the employee insertion but keep the department)

ROLLBACK TO SAVEPOINT savepoint1;

-- Commit the transaction (only the department insertion is saved)

COMMIT;

### ****e) User-Defined Stored Procedure:****

**Example:** Create a procedure to calculate the average salary of employees in a department:

DELIMITER $$

CREATE PROCEDURE GetAvgSalaryByDept(IN dept\_id INT)

BEGIN

SELECT AVG(emp\_salary) AS avg\_salary

FROM EMPLOYEES

WHERE dept\_no = dept\_id;

END$$

DELIMITER ;

**Call the Procedure:**

CALL GetAvgSalaryByDept(10); -- Replace 10 with the desired department number

**5.Create the following tables,**

**Event (eventid, name, description,city)**

**Participant (playerid, name, eventid, gender, year)**

**Prizes (prizeid, prize-money, eventid, rank,year)**

**Winners (prizeid, playerid)**

**a)Choose appropriate primary keys and foreign keys for the tables.**

**b)Playerid should contain at least one digit character.**

**c)Retrieve the name of events where all prize winners are females**

**d)Create a non-updatable view to retrieve the names of all participants who won 1st prizes along with their event names**

**e)Write a trigger to make sure that for every new event created, 3 prizes are created in prizes table. (1st prize - 1500, 2nd - 1000, 3rd 500)**

### Step-by-Step Explanation and Code:

**1. Create Tables with Primary Keys and Foreign Keys:**

-- Create Event table

CREATE TABLE Event (

eventid INT PRIMARY KEY,

name VARCHAR(100),

description TEXT,

city VARCHAR(100)

);

-- Create Participant table with CHECK constraint for playerid

CREATE TABLE Participant (

playerid VARCHAR(20) PRIMARY KEY,

name VARCHAR(100),

eventid INT,

gender ENUM('M', 'F'),

year INT,

FOREIGN KEY (eventid) REFERENCES Event(eventid),

CHECK (playerid REGEXP '[0-9]') -- At least one digit

);

-- Create Prizes table

CREATE TABLE Prizes (

prizeid INT PRIMARY KEY,

prize\_money DECIMAL(10, 2),

eventid INT,

rank INT,

year INT,

FOREIGN KEY (eventid) REFERENCES Event(eventid)

);

-- Create Winners table (junction table)

CREATE TABLE Winners (

prizeid INT,

playerid VARCHAR(20),

PRIMARY KEY (prizeid, playerid),

FOREIGN KEY (prizeid) REFERENCES Prizes(prizeid),

FOREIGN KEY (playerid) REFERENCES Participant(playerid)

);

### ****b) Playerid Must Contain at Least One Digit:****

The CHECK constraint in the Participant table ensures playerid has at least one digit:

CHECK (playerid REGEXP '[0-9]')

### ****c) Events Where All Prize Winners Are Female:****

SELECT e.name AS event\_name

FROM Event e

WHERE NOT EXISTS (

SELECT 1

FROM Prizes p

JOIN Winners w ON p.prizeid = w.prizeid

JOIN Participant pt ON w.playerid = pt.playerid

WHERE p.eventid = e.eventid

AND pt.gender = 'M' -- Exclude events with male winners

);

### ****d) Non-Updatable View for 1st Prize Winners:****

CREATE VIEW FirstPrizeWinners AS

SELECT p.name AS participant\_name, e.name AS event\_name

FROM Participant p

JOIN Winners w ON p.playerid = w.playerid

JOIN Prizes pr ON w.prizeid = pr.prizeid

JOIN Event e ON pr.eventid = e.eventid

WHERE pr.rank = 1;

### ****e) Trigger to Auto-Create Prizes for New Events:****

DELIMITER $$

CREATE TRIGGER after\_event\_insert

AFTER INSERT ON Event

FOR EACH ROW

BEGIN

-- Insert 3 prizes for the new event

INSERT INTO Prizes (prize\_money, eventid, rank, year)

VALUES

(1500, NEW.eventid, 1, YEAR(CURDATE())),

(1000, NEW.eventid, 2, YEAR(CURDATE())),

(500, NEW.eventid, 3, YEAR(CURDATE()));

END$$

DELIMITER ;

**6.Consider the schema for Movie Database:**

**ACTOR (Act\_id, Act\_Name, Act\_Gender) DIRECTOR (Dir\_id, Dir\_Name, Dir\_Phone)**

**MOVIES (Mov\_id, Mov\_Title, Mov\_Year, Mov\_Lang, Dir\_id) MOVIE\_CAST (Act\_id, Mov\_id, Role)**

**a)List the titles of all movies directed by ‘XXXX’.**

**b)Find the movie names where one or more actors acted in two or more movies.**

**c)List all actors who acted in a movie before 2010 and also in a movie after 2015 (use JOIN operation).**

**d)Create a view of movies with a particular actor with director.**

**e)Demonstrate the User defined function for the movie database.**

### Step-by-Step Explanation and Code:

**1. Create the Tables:**

-- Create ACTOR table

CREATE TABLE ACTOR (

Act\_id INT PRIMARY KEY,

Act\_Name VARCHAR(100),

Act\_Gender ENUM('M', 'F')

);

-- Create DIRECTOR table

CREATE TABLE DIRECTOR (

Dir\_id INT PRIMARY KEY,

Dir\_Name VARCHAR(100),

Dir\_Phone VARCHAR(15)

);

-- Create MOVIES table

CREATE TABLE MOVIES (

Mov\_id INT PRIMARY KEY,

Mov\_Title VARCHAR(100),

Mov\_Year YEAR,

Mov\_Lang VARCHAR(50),

Dir\_id INT,

FOREIGN KEY (Dir\_id) REFERENCES DIRECTOR(Dir\_id)

);

-- Create MOVIE\_CAST table

CREATE TABLE MOVIE\_CAST (

Act\_id INT,

Mov\_id INT,

Role VARCHAR(50),

PRIMARY KEY (Act\_id, Mov\_id),

FOREIGN KEY (Act\_id) REFERENCES ACTOR(Act\_id),

FOREIGN KEY (Mov\_id) REFERENCES MOVIES(Mov\_id)

);

### ****a) List Titles of Movies Directed by ‘XXXX’:****

SELECT m.Mov\_Title

FROM MOVIES m

JOIN DIRECTOR d ON m.Dir\_id = d.Dir\_id

WHERE d.Dir\_Name = 'XXXX';

### ****b) Find Movie Names Where Actors Acted in ≥2 Movies:****

SELECT DISTINCT m.Mov\_Title

FROM MOVIES m

JOIN MOVIE\_CAST mc ON m.Mov\_id = mc.Mov\_id

WHERE mc.Act\_id IN (

SELECT Act\_id

FROM MOVIE\_CAST

GROUP BY Act\_id

HAVING COUNT(Mov\_id) >= 2

);

### ****c) List Actors in Movies Before 2010 AND After 2015 (Using JOIN):****

SELECT DISTINCT a.Act\_Name

FROM ACTOR a

JOIN MOVIE\_CAST mc1 ON a.Act\_id = mc1.Act\_id

JOIN MOVIES m1 ON mc1.Mov\_id = m1.Mov\_id AND m1.Mov\_Year < 2010

JOIN MOVIE\_CAST mc2 ON a.Act\_id = mc2.Act\_id

JOIN MOVIES m2 ON mc2.Mov\_id = m2.Mov\_id AND m2.Mov\_Year > 2015;

### ****d) Create a View Linking Movies, Director, and a Specific Actor (e.g., 'John Doe'):****

CREATE VIEW MovieDirectorActorView AS

SELECT m.Mov\_Title, d.Dir\_Name, a.Act\_Name

FROM MOVIES m

JOIN DIRECTOR d ON m.Dir\_id = d.Dir\_id

JOIN MOVIE\_CAST mc ON m.Mov\_id = mc.Mov\_id

JOIN ACTOR a ON mc.Act\_id = a.Act\_id

WHERE a.Act\_Name = 'John Doe'; -- Replace with the desired actor name

### ****e) User-Defined Function to Count Movies by Director:****

DELIMITER $$

CREATE FUNCTION GetMovieCountByDirector(dir\_id INT)

RETURNS INT

DETERMINISTIC

BEGIN

DECLARE movie\_count INT;

SELECT COUNT(\*) INTO movie\_count

FROM MOVIES

WHERE Dir\_id = dir\_id;

RETURN movie\_count;

END$$

DELIMITER ;

**Usage Example:**

SELECT GetMovieCountByDirector(5); -- Returns the number of movies by director with Dir\_id=5

**7.Consider the schema for College Database:**

**STUDENT (RegNo, StudName, Address, Phone, Gender)**

**SUBJECT (Subcode, Title, Sem, Credits)**

**MARKS (RegNo, Subcode, Test1, Test2, Test3, Finalmark)**

**a)Compute the total number of male and female students in each semester and in each section.**

**b)Calculate the Finalmark (average of best two test marks) and update the corresponding table for all students.**

**c)Categorize students based on the following criterion: If Finalmark = 81 to 100 then CAT = ‘Outstanding’ If Finalmark = 51 to 80 then CAT = ‘Average’**

**If Finalmark < 51 then CAT = ‘Weak**

**d)Create a view of Test3 marks of particular student in all subjects.**

**e)Demonstrate the procedure for the above Database.**

### Step-by-Step Explanation and Code:

**1. Create the Tables:**

-- Create STUDENT table

CREATE TABLE STUDENT (

RegNo INT PRIMARY KEY,

StudName VARCHAR(100),

Address VARCHAR(200),

Phone VARCHAR(15),

Gender ENUM('M', 'F')

);

-- Create SUBJECT table

CREATE TABLE SUBJECT (

Subcode VARCHAR(10) PRIMARY KEY,

Title VARCHAR(100),

Sem INT,

Credits INT

);

-- Create MARKS table

CREATE TABLE MARKS (

RegNo INT,

Subcode VARCHAR(10),

Test1 DECIMAL(5,2),

Test2 DECIMAL(5,2),

Test3 DECIMAL(5,2),

Finalmark DECIMAL(5,2),

PRIMARY KEY (RegNo, Subcode),

FOREIGN KEY (RegNo) REFERENCES STUDENT(RegNo),

FOREIGN KEY (Subcode) REFERENCES SUBJECT(Subcode)

);

### ****a) Total Male/Female Students in Each Semester:****

SELECT

s.Sem,

st.Gender,

COUNT(DISTINCT st.RegNo) AS Total\_Students

FROM SUBJECT s

JOIN MARKS m ON s.Subcode = m.Subcode

JOIN STUDENT st ON m.RegNo = st.RegNo

GROUP BY s.Sem, st.Gender;

### ****b) Update****Finalmark****as Average of Best Two Tests:****

UPDATE MARKS

SET Finalmark = (

(Test1 + Test2 + Test3 - LEAST(Test1, Test2, Test3)) / 2

);

### ****c) Categorize Students Based on****Finalmark****:****

SELECT

RegNo,

Subcode,

Finalmark,

CASE

WHEN Finalmark BETWEEN 81 AND 100 THEN 'Outstanding'

WHEN Finalmark BETWEEN 51 AND 80 THEN 'Average'

ELSE 'Weak'

END AS CAT

FROM MARKS;

### ****d) View for Test3 Marks of a Specific Student:****

CREATE VIEW StudentTest3View AS

SELECT

st.StudName,

s.Title AS Subject,

m.Test3

FROM MARKS m

JOIN STUDENT st ON m.RegNo = st.RegNo

JOIN SUBJECT s ON m.Subcode = s.Subcode;

-- Example usage for student with RegNo=101:

SELECT \* FROM StudentTest3View WHERE RegNo = 101;

### ****e) Stored Procedure to Add a New Student:****

DELIMITER $$

CREATE PROCEDURE AddStudent(

IN p\_RegNo INT,

IN p\_StudName VARCHAR(100),

IN p\_Address VARCHAR(200),

IN p\_Phone VARCHAR(15),

IN p\_Gender ENUM('M', 'F')

)

BEGIN

INSERT INTO STUDENT (RegNo, StudName, Address, Phone, Gender)

VALUES (p\_RegNo, p\_StudName, p\_Address, p\_Phone, p\_Gender);

END$$

DELIMITER ;

-- Usage:

CALL AddStudent(101, 'John Doe', '123 Main St', '555-1234', 'M');

**8.Create table as Bank ( S.No,Cust\_Name, Acc\_No, Balance, Branch),**

**a)Select with where clause.**

**b)Select with comparison operator.**

**c)Update the balance in the second row.**

**d)Select with between in the field balance.**

**e)Write a trigger when balance is below 1000.**

-- Create Bank table

CREATE TABLE Bank (

S.No INT PRIMARY KEY AUTO\_INCREMENT,

Cust\_Name VARCHAR(50) NOT NULL,

Acc\_No VARCHAR(20) UNIQUE NOT NULL,

Balance DECIMAL(15,2) DEFAULT 0.00,

Branch VARCHAR(50)

);

-- a) Select with WHERE clause

SELECT Cust\_Name, Acc\_No, Balance

FROM Bank

WHERE Branch = 'Main';

-- b) Select with comparison operator

SELECT \*

FROM Bank

WHERE Balance > 5000;

-- c) Update balance in second row (assuming S.No=2)

UPDATE Bank

SET Balance = 7500.00

WHERE S.No = 2;

-- d) Select with BETWEEN for balance

SELECT Cust\_Name, Acc\_No

FROM Bank

WHERE Balance BETWEEN 1000 AND 10000;

-- e) Create trigger for low balance

-- First create notifications table

CREATE TABLE low\_balance\_notifications (

id INT PRIMARY KEY AUTO\_INCREMENT,

acc\_no VARCHAR(20),

balance DECIMAL(15,2),

notification\_date DATETIME

);

DELIMITER //

CREATE TRIGGER check\_balance

AFTER UPDATE ON Bank

FOR EACH ROW

BEGIN

IF NEW.Balance < 1000 THEN

INSERT INTO low\_balance\_notifications (acc\_no, balance, notification\_date)

VALUES (NEW.Acc\_No, NEW.Balance, NOW());

END IF;

END; //

DELIMITER ;

**9.Create a table Account (Account\_No, Cust\_Name, Branch\_Name, Account\_Balance, Account\_Type) Select an appropriate primary key.**

**a)Display the Cust\_Name and Account\_No of the customers of "Branch = XXXXX".**

**b)Display the names and account types of all the customers whose account balance is more than 10,000.**

**c)Add column Cust\_Date\_of Birth in the ACCOUNT table.**

**d)Display Account\_No, Cust\_Name and Branch of all the customers whose account balance is less than 1,000.**

**e)Write a procedure for the above Database.**

-- Create Account table

CREATE TABLE Account (

Account\_No VARCHAR(20) PRIMARY KEY,

Cust\_Name VARCHAR(50) NOT NULL,

Branch\_Name VARCHAR(50),

Account\_Balance DECIMAL(15,2) DEFAULT 0.00,

Account\_Type VARCHAR(20)

);

-- a) Display customers of specific branch

SELECT Cust\_Name, Account\_No

FROM Account

WHERE Branch\_Name = 'XXXXX';

-- b) High balance accounts

SELECT Cust\_Name, Account\_Type

FROM Account

WHERE Account\_Balance > 10000;

-- c) Add Date of Birth column

ALTER TABLE Account

ADD COLUMN Cust\_Date\_of\_Birth DATE;

-- d) Low balance accounts

SELECT Account\_No, Cust\_Name, Branch\_Name

FROM Account

WHERE Account\_Balance < 1000;

-- e) Create stored procedure

DELIMITER //

CREATE PROCEDURE GetCustomerDetails(

IN branch\_name VARCHAR(50),

IN min\_balance DECIMAL(15,2)

)

BEGIN

SELECT Account\_No, Cust\_Name, Branch\_Name, Account\_Balance, Account\_Type

FROM Account

WHERE Branch\_Name = branch\_name

AND Account\_Balance >= min\_balance;

END //

DELIMITER ;

**10. Create the tables CUSTOMER(C\_ID,Name,Address,City,Mobile\_No) and ORDER (C\_ID, P\_ID, P\_Name, P\_COST),**

**a)List the names and addresses of all the customers who have ordered products of costmore than 500.**

**b)List the names of all the products ordered whose cost is 1,000 or more.**

**c)List the product names which are ordered by customers of "City = Delhi".**

**d)Add column "Email\_id" in the CUSTOMER table.**

**e)Demonstrate the user defined function for the above tables.**

-- Create tables

CREATE TABLE CUSTOMER (

C\_ID INT PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

Address VARCHAR(100),

City VARCHAR(50),

Mobile\_No VARCHAR(15)

);

CREATE TABLE `ORDER` (

C\_ID INT,

P\_ID INT,

P\_Name VARCHAR(50),

P\_COST DECIMAL(10,2),

PRIMARY KEY (C\_ID, P\_ID),

FOREIGN KEY (C\_ID) REFERENCES CUSTOMER(C\_ID)

);

-- a) Customers with orders > 500

SELECT DISTINCT C.Name, C.Address

FROM CUSTOMER C

JOIN `ORDER` O ON C.C\_ID = O.C\_ID

WHERE O.P\_COST > 500;

-- b) Products costing ≥1000

SELECT DISTINCT P\_Name

FROM `ORDER`

WHERE P\_COST >= 1000;

-- c) Products ordered from Delhi

SELECT DISTINCT O.P\_Name

FROM `ORDER` O

JOIN CUSTOMER C ON O.C\_ID = C.C\_ID

WHERE C.City = 'Delhi';

-- d) Add Email column

ALTER TABLE CUSTOMER

ADD COLUMN Email\_id VARCHAR(100);

-- e) User-defined function (total spent by customer)

DELIMITER //

CREATE FUNCTION GetTotalSpent(customer\_id INT)

RETURNS DECIMAL(10,2)

DETERMINISTIC

BEGIN

DECLARE total DECIMAL(10,2);

SELECT COALESCE(SUM(P\_COST), 0) INTO total

FROM `ORDER`

WHERE C\_ID = customer\_id;

RETURN total;

END //

DELIMITER ;

-- Example usage of the function:

SELECT

C\_ID,

Name,

GetTotalSpent(C\_ID) AS Total\_Spent

FROM CUSTOMER;

**11.Create the tables**

**SALESMAN (Salesman\_id, Name, City,Commission),**

**CUSTOMER (Customer\_id, Cust\_Name, City, Grade,Salesman\_id),**

**ORDERS (Ord\_No, Purchase\_Amt, Ord\_Date, Customer\_id, Salesman\_id)**

**a)Find the name and numbers of all salesmen who had more than one customer.**

**b)List all salesmen and indicate those who have and don’t have customers in their cities (Use UNION operation.)**

**c)Create a view that finds the salesman who has the customer with the highest order of a day.**

**d)Perform the DELETE operation by removing salesman with id 1000. All his orders must also be deleted.**

**e)Demonstrate the Triggers for the above table.**

-- Create tables with proper constraints

CREATE TABLE SALESMAN (

Salesman\_id INT PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

City VARCHAR(50),

Commission DECIMAL(10,2) DEFAULT 0

);

CREATE TABLE CUSTOMER (

Customer\_id INT PRIMARY KEY,

Cust\_Name VARCHAR(50) NOT NULL,

City VARCHAR(50),

Grade INT,

Salesman\_id INT,

FOREIGN KEY (Salesman\_id) REFERENCES SALESMAN(Salesman\_id) ON DELETE SET NULL

);

CREATE TABLE ORDERS (

Ord\_No INT PRIMARY KEY,

Purchase\_Amt DECIMAL(10,2) NOT NULL,

Ord\_Date DATE NOT NULL,

Customer\_id INT,

Salesman\_id INT,

FOREIGN KEY (Customer\_id) REFERENCES CUSTOMER(Customer\_id),

FOREIGN KEY (Salesman\_id) REFERENCES SALESMAN(Salesman\_id) ON DELETE CASCADE

);

-- a) Salesmen with multiple customers

SELECT S.Salesman\_id, S.Name, S.City, COUNT(C.Customer\_id) AS Customer\_Count

FROM SALESMAN S

JOIN CUSTOMER C ON S.Salesman\_id = C.Salesman\_id

GROUP BY S.Salesman\_id

HAVING COUNT(C.Customer\_id) > 1;

-- b) Salesmen with/without local customers using UNION

SELECT S.Salesman\_id, S.Name, 'Has Customer in City' AS Status

FROM SALESMAN S

WHERE EXISTS (

SELECT 1

FROM CUSTOMER C

WHERE C.Salesman\_id = S.Salesman\_id

AND C.City = S.City

)

UNION

SELECT S.Salesman\_id, S.Name, 'No Customer in City'

FROM SALESMAN S

WHERE NOT EXISTS (

SELECT 1

FROM CUSTOMER C

WHERE C.Salesman\_id = S.Salesman\_id

AND C.City = S.City

);

-- c) View for highest daily order salesman

CREATE VIEW TopDailySalesman AS

SELECT

O.Ord\_Date,

S.Salesman\_id,

S.Name,

MAX(O.Purchase\_Amt) AS Max\_Purchase

FROM ORDERS O

JOIN SALESMAN S ON O.Salesman\_id = S.Salesman\_id

GROUP BY O.Ord\_Date;

-- d) Delete salesman 1000 and orders

DELETE FROM SALESMAN WHERE Salesman\_id = 1000;

-- e) Triggers

-- Trigger 1: Update commission on new order

DELIMITER //

CREATE TRIGGER UpdateCommission

AFTER INSERT ON ORDERS

FOR EACH ROW

BEGIN

UPDATE SALESMAN

SET Commission = Commission + (NEW.Purchase\_Amt \* 0.05)

WHERE Salesman\_id = NEW.Salesman\_id;

END//

DELIMITER ;

-- Trigger 2: Prevent negative purchase amounts

DELIMITER //

CREATE TRIGGER ValidatePurchaseAmount

BEFORE INSERT ON ORDERS

FOR EACH ROW

BEGIN

IF NEW.Purchase\_Amt < 0 THEN

SIGNAL SQLSTATE '45000'

SET MESSAGE\_TEXT = 'Purchase amount cannot be negative';

END IF;

END//

DELIMITER ;

**15.Create a Employee database and write SQL Triggers for insert, delete, and update operations in a database table.**

## **1. Create the Database and Table**

-- Create the database

CREATE DATABASE IF NOT EXISTS EmployeeDB;

USE EmployeeDB;

-- Create the Employees table

CREATE TABLE IF NOT EXISTS Employees (

employee\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

department VARCHAR(50),

salary DECIMAL(10, 2),

hire\_date DATE NOT NULL,

last\_updated TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

);

-- Create an audit log table to track changes

CREATE TABLE IF NOT EXISTS EmployeeAuditLog (

log\_id INT AUTO\_INCREMENT PRIMARY KEY,

employee\_id INT,

action\_type VARCHAR(10) NOT NULL, -- INSERT, UPDATE, DELETE

action\_timestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

old\_data JSON,

new\_data JSON,

changed\_by VARCHAR(50) DEFAULT USER()

);

## **2. Create Triggers**

### INSERT Trigger

This trigger logs all new employee additions:

DELIMITER //

CREATE TRIGGER after\_employee\_insert

AFTER INSERT ON Employees

FOR EACH ROW

BEGIN

INSERT INTO EmployeeAuditLog (

employee\_id,

action\_type,

new\_data

) VALUES (

NEW.employee\_id,

'INSERT',

JSON\_OBJECT(

'first\_name', NEW.first\_name,

'last\_name', NEW.last\_name,

'email', NEW.email,

'department', NEW.department,

'salary', NEW.salary,

'hire\_date', NEW.hire\_date

)

);

END //

DELIMITER ;

### UPDATE Trigger

This trigger logs changes to employee records:

DELIMITER //

CREATE TRIGGER before\_employee\_update

BEFORE UPDATE ON Employees

FOR EACH ROW

BEGIN

INSERT INTO EmployeeAuditLog (

employee\_id,

action\_type,

old\_data,

new\_data

) VALUES (

NEW.employee\_id,

'UPDATE',

JSON\_OBJECT(

'first\_name', OLD.first\_name,

'last\_name', OLD.last\_name,

'email', OLD.email,

'department', OLD.department,

'salary', OLD.salary,

'hire\_date', OLD.hire\_date

),

JSON\_OBJECT(

'first\_name', NEW.first\_name,

'last\_name', NEW.last\_name,

'email', NEW.email,

'department', NEW.department,

'salary', NEW.salary,

'hire\_date', NEW.hire\_date

)

);

END //

DELIMITER ;

### DELETE Trigger

This trigger logs when employees are removed:

DELIMITER //

CREATE TRIGGER before\_employee\_delete

BEFORE DELETE ON Employees

FOR EACH ROW

BEGIN

INSERT INTO EmployeeAuditLog (

employee\_id,

action\_type,

old\_data

) VALUES (

OLD.employee\_id,

'DELETE',

JSON\_OBJECT(

'first\_name', OLD.first\_name,

'last\_name', OLD.last\_name,

'email', OLD.email,

'department', OLD.department,

'salary', OLD.salary,

'hire\_date', OLD.hire\_date

)

);

END //

DELIMITER ;

## **3. Test the Triggers**

Let's test the triggers with some sample operations:

-- Insert a new employee

INSERT INTO Employees (first\_name, last\_name, email, department, salary, hire\_date)

VALUES ('John', 'Doe', 'john.doe@example.com', 'IT', 75000.00, '2023-01-15');

-- Update the employee

UPDATE Employees

SET salary = 80000.00, department = 'Engineering'

WHERE employee\_id = 1;

-- Delete the employee

DELETE FROM Employees WHERE employee\_id = 1;

-- View the audit log

SELECT \* FROM EmployeeAuditLog;

To drop a trigger:

DROP TRIGGER IF EXISTS before\_employee\_insert;

**16.Create a table Supplier (Sup\_No, Sup\_Name ,Item\_Supplied , Item\_Price , City)**

**a)Write sql query to display Suplier numbers and Supplier names whose name starts with ‘S’**

**b)Write sql query to add a new column called CONTACTNO.**

**c)Write sql query to display supplier numbers, Suplier names and item price for suppliers in Chennai in the ascending order of item price.**

**d)Create a view on the table which displays only supplier numbers and supplier names.**

**e)Demonstrate the procedure for the supplier table.**

-- Create Supplier table

CREATE TABLE Supplier (

Sup\_No INT PRIMARY KEY,

Sup\_Name VARCHAR(50) NOT NULL,

Item\_Supplied VARCHAR(50),

Item\_Price DECIMAL(10,2),

City VARCHAR(50)

);

-- a) Suppliers with names starting with 'S'

SELECT Sup\_No, Sup\_Name

FROM Supplier

WHERE Sup\_Name LIKE 'S%';

-- b) Add contact number column

ALTER TABLE Supplier

ADD COLUMN CONTACTNO VARCHAR(15);

-- c) Chennai suppliers sorted by price

SELECT Sup\_No, Sup\_Name, Item\_Price

FROM Supplier

WHERE City = 'Chennai'

ORDER BY Item\_Price ASC;

-- d) Create supplier view

CREATE VIEW SupplierBasicInfo AS

SELECT Sup\_No, Sup\_Name

FROM Supplier;

-- e) Stored procedure demonstration

DELIMITER //

CREATE PROCEDURE GetSuppliers()

BEGIN

SELECT \* FROM Supplier;

END //

DELIMITER ;

**12.Develop a simple GUI based Inventory Management for a EMart Grocery Shop database application and incorporate all the Database features.**

Refer lab manual

**13.** **Create an XML database for the student profile and validate it using XML schema.**

## **1. MySQL Database Setup**

First, let's create a MySQL database schema for student profiles:

CREATE DATABASE IF NOT EXISTS StudentDB;

USE StudentDB;

CREATE TABLE IF NOT EXISTS Students (

student\_id INT AUTO\_INCREMENT PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

date\_of\_birth DATE NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

major VARCHAR(50),

enrollment\_date DATE NOT NULL,

gpa DECIMAL(3,2),

is\_active BOOLEAN DEFAULT TRUE

);

CREATE TABLE IF NOT EXISTS Courses (

course\_id INT AUTO\_INCREMENT PRIMARY KEY,

course\_code VARCHAR(20) NOT NULL,

course\_name VARCHAR(100) NOT NULL,

credits INT NOT NULL

);

CREATE TABLE IF NOT EXISTS StudentCourses (

enrollment\_id INT AUTO\_INCREMENT PRIMARY KEY,

student\_id INT NOT NULL,

course\_id INT NOT NULL,

enrollment\_date DATE NOT NULL,

grade VARCHAR(2),

FOREIGN KEY (student\_id) REFERENCES Students(student\_id),

FOREIGN KEY (course\_id) REFERENCES Courses(course\_id)

);

## **2. XML Schema Definition (XSD)**

Create a file student\_schema.xsd to validate student profiles:

<?xml version="1.0" encoding="UTF-8"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="studentProfile">

<xs:complexType>

<xs:sequence>

<xs:element name="student" maxOccurs="unbounded">

<xs:complexType>

<xs:sequence>

<xs:element name="firstName" type="xs:string"/>

<xs:element name="lastName" type="xs:string"/>

<xs:element name="dateOfBirth" type="xs:date"/>

<xs:element name="email" type="xs:string"/>

<xs:element name="major" type="xs:string" minOccurs="0"/>

<xs:element name="enrollmentDate" type="xs:date"/>

<xs:element name="gpa" type="xs:decimal" minOccurs="0"/>

<xs:element name="isActive" type="xs:boolean" default="true"/>

<xs:element name="courses" minOccurs="0">

<xs:complexType>

<xs:sequence>

<xs:element name="course" maxOccurs="unbounded">

<xs:complexType>

<xs:sequence>

<xs:element name="courseCode" type="xs:string"/>

<xs:element name="courseName" type="xs:string"/>

<xs:element name="credits" type="xs:positiveInteger"/>

<xs:element name="enrollmentDate" type="xs:date"/>

<xs:element name="grade" type="xs:string" minOccurs="0"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:sequence>

<xs:attribute name="studentID" type="xs:positiveInteger"/>

</xs:complexType>

</xs:element>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

## **3. Python Implementation**

Create a Python script student\_xml\_db.py with the following functionality:

import mysql.connector

from lxml import etree

from xml.etree import ElementTree as ET

from datetime import datetime

class StudentXMLDatabase:

def \_\_init\_\_(self):

# Database connection

self.db = mysql.connector.connect(

host="localhost",

user="your\_username",

password="your\_password",

database="StudentDB"

)

self.cursor = self.db.cursor()

# Load XML schema for validation

self.schema = etree.XMLSchema(etree.parse('student\_schema.xsd'))

def validate\_xml(self, xml\_file):

"""Validate XML against the schema"""

try:

xml\_doc = etree.parse(xml\_file)

self.schema.assertValid(xml\_doc)

print("XML is valid!")

return True

except etree.DocumentInvalid as e:

print("XML validation failed:", e)

return False

except Exception as e:

print("Error:", e)

return False

def xml\_to\_mysql(self, xml\_file):

"""Insert validated XML data into MySQL"""

if not self.validate\_xml(xml\_file):

return False

try:

tree = ET.parse(xml\_file)

root = tree.getroot()

for student in root.findall('student'):

# Extract student data

student\_id = student.get('studentID')

first\_name = student.find('firstName').text

last\_name = student.find('lastName').text

dob = student.find('dateOfBirth').text

email = student.find('email').text

major = student.find('major').text if student.find('major') is not None else None

enrollment\_date = student.find('enrollmentDate').text

gpa = float(student.find('gpa').text) if student.find('gpa') is not None else None

is\_active = student.find('isActive').text.lower() == 'true'

# Insert student

if student\_id:

sql = """INSERT INTO Students

(student\_id, first\_name, last\_name, date\_of\_birth, email,

major, enrollment\_date, gpa, is\_active)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s)"""

val = (student\_id, first\_name, last\_name, dob, email,

major, enrollment\_date, gpa, is\_active)

else:

sql = """INSERT INTO Students

(first\_name, last\_name, date\_of\_birth, email,

major, enrollment\_date, gpa, is\_active)

VALUES (%s, %s, %s, %s, %s, %s, %s, %s)"""

val = (first\_name, last\_name, dob, email,

major, enrollment\_date, gpa, is\_active)

self.cursor.execute(sql, val)

student\_db\_id = self.cursor.lastrowid if not student\_id else student\_id

# Process courses if they exist

courses = student.find('courses')

if courses is not None:

for course in courses.findall('course'):

course\_code = course.find('courseCode').text

course\_name = course.find('courseName').text

credits = int(course.find('credits').text)

course\_enrollment\_date = course.find('enrollmentDate').text

grade = course.find('grade').text if course.find('grade') is not None else None

# Check if course exists

self.cursor.execute(

"SELECT course\_id FROM Courses WHERE course\_code = %s",

(course\_code,)

)

result = self.cursor.fetchone()

if result:

course\_id = result[0]

else:

# Insert new course

self.cursor.execute(

"""INSERT INTO Courses

(course\_code, course\_name, credits)

VALUES (%s, %s, %s)""",

(course\_code, course\_name, credits)

)

course\_id = self.cursor.lastrowid

# Enroll student in course

self.cursor.execute(

"""INSERT INTO StudentCourses

(student\_id, course\_id, enrollment\_date, grade)

VALUES (%s, %s, %s, %s)""",

(student\_db\_id, course\_id, course\_enrollment\_date, grade)

)

self.db.commit()

print("Data successfully imported to MySQL")

return True

except Exception as e:

self.db.rollback()

print("Error importing to MySQL:", e)

return False

def mysql\_to\_xml(self, output\_file):

"""Export MySQL data to XML"""

try:

# Create root element

root = ET.Element("studentProfile")

# Get all students

self.cursor.execute("SELECT \* FROM Students")

students = self.cursor.fetchall()

for student in students:

student\_elem = ET.SubElement(root, "student")

student\_elem.set("studentID", str(student[0]))

ET.SubElement(student\_elem, "firstName").text = student[1]

ET.SubElement(student\_elem, "lastName").text = student[2]

ET.SubElement(student\_elem, "dateOfBirth").text = student[3].strftime('%Y-%m-%d')

ET.SubElement(student\_elem, "email").text = student[4]

if student[5]:

ET.SubElement(student\_elem, "major").text = student[5]

ET.SubElement(student\_elem, "enrollmentDate").text = student[6].strftime('%Y-%m-%d')

if student[7]:

ET.SubElement(student\_elem, "gpa").text = str(student[7])

ET.SubElement(student\_elem, "isActive").text = 'true' if student[8] else 'false'

# Get student's courses

self.cursor.execute(

"""SELECT c.course\_code, c.course\_name, c.credits,

sc.enrollment\_date, sc.grade

FROM StudentCourses sc

JOIN Courses c ON sc.course\_id = c.course\_id

WHERE sc.student\_id = %s""",

(student[0],)

)

courses = self.cursor.fetchall()

if courses:

courses\_elem = ET.SubElement(student\_elem, "courses")

for course in courses:

course\_elem = ET.SubElement(courses\_elem, "course")

ET.SubElement(course\_elem, "courseCode").text = course[0]

ET.SubElement(course\_elem, "courseName").text = course[1]

ET.SubElement(course\_elem, "credits").text = str(course[2])

ET.SubElement(course\_elem, "enrollmentDate").text = course[3].strftime('%Y-%m-%d')

if course[4]:

ET.SubElement(course\_elem, "grade").text = course[4]

# Write to file

tree = ET.ElementTree(root)

tree.write(output\_file, encoding='utf-8', xml\_declaration=True)

print(f"Data successfully exported to {output\_file}")

# Validate the generated XML

self.validate\_xml(output\_file)

return True

except Exception as e:

print("Error exporting to XML:", e)

return False

def close(self):

"""Close database connection"""

self.cursor.close()

self.db.close()

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

db = StudentXMLDatabase()

# Example XML file to import

sample\_xml = """

<studentProfile>

<student studentID="101">

<firstName>John</firstName>

<lastName>Doe</lastName>

<dateOfBirth>2000-05-15</dateOfBirth>

<email>john.doe@university.edu</email>

<major>Computer Science</major>

<enrollmentDate>2022-09-01</enrollmentDate>

<gpa>3.75</gpa>

<isActive>true</isActive>

<courses>

<course>

<courseCode>CS101</courseCode>

<courseName>Introduction to Programming</courseName>

<credits>4</credits>

<enrollmentDate>2022-09-01</enrollmentDate>

<grade>A</grade>

</course>

<course>

<courseCode>MATH201</courseCode>

<courseName>Calculus I</courseName>

<credits>3</credits>

<enrollmentDate>2022-09-01</enrollmentDate>

<grade>B+</grade>

</course>

</courses>

</student>

<student>

<firstName>Jane</firstName>

<lastName>Smith</lastName>

<dateOfBirth>2001-02-20</dateOfBirth>

<email>jane.smith@university.edu</email>

<major>Biology</major>

<enrollmentDate>2023-01-15</enrollmentDate>

<gpa>3.9</gpa>

<isActive>true</isActive>

</student>

</studentProfile>

"""

# Write sample XML to file

with open('sample\_students.xml', 'w') as f:

f.write(sample\_xml)

# Import XML to MySQL

db.xml\_to\_mysql('sample\_students.xml')

# Export MySQL to XML

db.mysql\_to\_xml('exported\_students.xml')

db.close()

## **4. How It Works**

1. **Database Schema**: The MySQL database has tables for Students, Courses, and Student-Course relationships.
2. **XML Schema (XSD)**: Defines the structure and validation rules for student profile XML files.
3. **Python Class**:
   * validate\_xml(): Validates XML against the XSD schema
   * xml\_to\_mysql(): Imports validated XML data into MySQL
   * mysql\_to\_xml(): Exports MySQL data to XML format
   * Handles all relationships between students and courses
4. **Example Usage**:
   * Creates a sample XML file
   * Imports it to MySQL
   * Exports the data back to XML
   * Validates at each step

## **5. Requirements**

To run this solution, you'll need:

* Python 3.x
* MySQL Server
* Python packages: mysql-connector-python, lxml

Install the packages with:

Copy

Download

pip install mysql-connector-python lxml

This implementation provides a complete solution for managing student profiles in both XML and MySQL formats with proper validation.