SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA 1st semester
DATABASE TECHNOLOGY (PCA20C03J)

Lab Manual

Lab 1: Case study submission for ER Diagrams

Title: ER Diagram Case Study Submission

Aim: To understand and apply the concepts of Entity-Relationship (ER) modeling by designing an ER diagram for a given case study, identifying entities, attributes, and relationships.

Procedure:

1. **Understand the Case Study:** Carefully read and analyze the provided case study description to identify the key business rules, entities, and their interactions.
2. **Identify Entities:** List all significant objects or concepts that need to store data (e.g., Student, Course, Instructor, Department).
3. **Identify Attributes:** For each identified entity, list its relevant properties or characteristics (e.g., for Student: StudentID, Name, DateOfBirth, Major).
4. **Identify Relationships:** Determine how entities are related to each other. Specify the type of relationship (one-to-one, one-to-many, many-to-many) and their cardinality (e.g., a Student *enrolls in* many Courses, a Course *is taught by* one Instructor).
5. **Draw the ER Diagram:** Use standard ER notation (e.g., Chen's notation, Crow's Foot notation) to visually represent the entities, attributes, and relationships. Include primary keys, foreign keys, and any necessary constraints.
6. **Review and Refine:** Check the ER diagram for accuracy, completeness, and adherence to normalization principles (if applicable). Ensure it accurately reflects the case study requirements.

Source Code:

- ER diagrams are conceptual designs and do not have "source code" in the traditional programming sense.
- Tools commonly used for drawing ER diagrams include:
    - Lucidchart
    - draw.io (now diagrams.net)
    - Microsoft Visio
    - MySQL Workbench (for database-specific ERDs)
- The "source" would be the conceptual design process itself and the chosen notation.

Input: A detailed textual description of a business scenario or system for which a database needs to be designed.

Expected Output: A clear, well-structured Entity-Relationship Diagram (ERD) that accurately models the data requirements of the given case study. The ERD should include:

- All identified entities.
- All relevant attributes for each entity, with primary keys underlined.
- Relationships between entities, indicating cardinality and participation constraints.
- Any necessary foreign keys.

Lab 2: SQL queries for students database

Title: SQL Queries for Student Database

Aim: To practice writing various SQL queries to retrieve, insert, update, and delete data from a student database, demonstrating fundamental DDL and DML operations.

Procedure:

1. **Connect to Database:** Establish a connection to your SQL database server (e.g., MySQL, PostgreSQL, SQL Server, Oracle).
2. **Create Database and Table:** Execute SQL DDL (Data Definition Language) commands to create a database (if needed) and a `Students` table with appropriate columns and data types.
3. **Insert Data:** Use SQL DML (Data Manipulation Language) `INSERT` statements to populate the `Students` table with sample data.
4. **Execute SELECT Queries:** Write and execute various `SELECT` queries to retrieve data based on different conditions (e.g., all students, students from a specific major, students older than a certain age).
5. **Execute UPDATE Queries:** Write and execute `UPDATE` queries to modify existing data in the `Students` table.
6. **Execute DELETE Queries:** Write and execute `DELETE` queries to remove data from the `Students` table.
7. **Verify Results:** After each DML operation, use `SELECT` to verify that the changes have been applied correctly.

Source Code:

```
-- DDL: Create a Students table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DateOfBirth DATE,
    Major VARCHAR(100),
    Email VARCHAR(100) UNIQUE
);

-- DML: Insert sample data
INSERT INTO Students (StudentID, FirstName, LastName, DateOfBirth, Major, Email)
VALUES
(101, 'Alice', 'Smith', '2003-05-15', 'Computer Science',
'alice.s@example.com'),
(102, 'Bob', 'Johnson', '2002-11-22', 'Electrical Engineering',
'bob.j@example.com'),
(103, 'Charlie', 'Brown', '2004-01-30', 'Computer Science',
'charlie.b@example.com'),
(104, 'Diana', 'Prince', '2003-07-01', 'Mathematics', 'diana.p@example.com');

-- DML: Select all students
SELECT * FROM Students;

-- DML: Select students majoring in 'Computer Science'
SELECT StudentID, FirstName, LastName FROM Students WHERE Major = 'Computer
Science';

-- DML: Select students born before 2003
```

```
SELECT FirstName, LastName, DateOfBirth FROM Students WHERE DateOfBirth < '2003-
01-01';

-- DML: Update a student's major
UPDATE Students SET Major = 'Software Engineering' WHERE StudentID = 101;

-- DML: Delete a student
DELETE FROM Students WHERE StudentID = 104;

-- DML: Verify changes (after update/delete)
SELECT * FROM Students;

-- Optional: Drop table (for cleanup)
-- DROP TABLE Students;
```

Input: The SQL queries themselves, along with the sample data provided in the `INSERT` statements.

Expected Output:

After initial `INSERT` and `SELECT *`: | StudentID | FirstName | LastName | DateOfBirth | Major | Email | | :-------- | :-------- | :------- | :---------- | :------------------- | :------------------ | | 101 | Alice | Smith | 2003-05-15 | Computer Science | alice.s@example.com | | 102 | Bob | Johnson | 2002-11-22 | Electrical Engineering | bob.j@example.com | | 103 | Charlie | Brown | 2004-01-30 | Computer Science | charlie.b@example.com | | 104 | Diana | Prince | 2003-07-01 | Mathematics | diana.p@example.com |

After `SELECT WHERE Major = 'Computer Science'`: | StudentID | FirstName | LastName | | :-------- | :-------- | :------- | | | 101 | Alice | Smith | | 103 | Charlie | Brown |

After `UPDATE` and final `SELECT *`: | StudentID | FirstName | LastName | DateOfBirth | Major | Email | | :-------- | :-------- | :------- | :---------- | :------------------- | :------------------ | | 101 | Alice | Smith | 2003-05-15 | Software Engineering | alice.s@example.com | | 102 | Bob | Johnson | 2002-11-22 | Electrical Engineering | bob.j@example.com | | 103 | Charlie | Brown | 2004-01-30 | Computer Science | charlie.b@example.com |

After `DELETE` and final `SELECT *`: | StudentID | FirstName | LastName | DateOfBirth | Major | Email | | :-------- | :-------- | :------- | :---------- | :------------------- | :------------------ | | 101 | Alice | Smith | 2003-05-15 | Software Engineering | alice.s@example.com | | 102 | Bob | Johnson | 2002-11-22 | Electrical Engineering | bob.j@example.com | | 103 | Charlie | Brown | 2004-01-30 | Computer Science | charlie.b@example.com |

Lab 3: SQL queries for employee database

Title: SQL Queries for Employee Database

Aim: To practice writing various SQL queries to manage data in an employee database, including creating tables, inserting records, and performing common data retrieval and modification operations.

Procedure:

1. **Connect to Database:** Establish a connection to your SQL database server.
2. **Create Database and Table:** Execute SQL DDL commands to create an `Employees` table with columns like `EmployeeID`, `FirstName`, `LastName`, `Department`, `Salary`, and `HireDate`.
3. **Insert Data:** Use `INSERT` statements to populate the `Employees` table with diverse sample data.
4. **Execute SELECT Queries:** Write and execute `SELECT` queries to retrieve employee information based on various criteria (e.g., employees in a specific department, employees with salary above a certain amount, employees hired after a specific date).
5. **Execute UPDATE Queries:** Write and execute `UPDATE` queries to modify employee details (e.g., change department, update salary).
6. **Execute DELETE Queries:** Write and execute `DELETE` queries to remove employee records.
7. **Verify Results:** Use `SELECT` statements to confirm the effects of DML operations.

Source Code:

```
-- DDL: Create an Employees table
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Department VARCHAR(100),
    Salary DECIMAL(10, 2),
    HireDate DATE
);

-- DML: Insert sample data
INSERT INTO Employees (EmployeeID, FirstName, LastName, Department, Salary,
HireDate) VALUES
(1, 'John', 'Doe', 'Sales', 60000.00, '2020-01-15'),
(2, 'Jane', 'Smith', 'Marketing', 75000.00, '2019-03-20'),
(3, 'Peter', 'Jones', 'Sales', 62000.00, '2021-06-10'),
(4, 'Mary', 'Brown', 'HR', 55000.00, '2022-02-01'),
(5, 'David', 'Lee', 'IT', 80000.00, '2018-09-01');

-- DML: Select all employees
SELECT * FROM Employees;

-- DML: Select employees in 'Sales' department
SELECT FirstName, LastName, Salary FROM Employees WHERE Department = 'Sales';

-- DML: Select employees with salary > 70000
SELECT FirstName, LastName, Salary FROM Employees WHERE Salary > 70000.00;

-- DML: Update an employee's department
UPDATE Employees SET Department = 'IT' WHERE EmployeeID = 3;
```

```
-- DML: Delete an employee
DELETE FROM Employees WHERE EmployeeID = 4;

-- DML: Verify changes (after update/delete)
SELECT * FROM Employees;

-- Optional: Drop table (for cleanup)
-- DROP TABLE Employees;
```

Input: The SQL queries themselves, along with the sample data provided in the `INSERT` statements.

Expected Output:

After initial `INSERT` and `SELECT *`: | EmployeeID | FirstName | LastName | Department | Salary | HireDate | | :--------- | :-------- | :------- | :--------- | :------- | :--------- | | 1 | John | Doe | Sales | 60000.00 | 2020-01-15 | | 2 | Jane | Smith | Marketing | 75000.00 | 2019-03-20 | | 3 | Peter | Jones | Sales | 62000.00 | 2021-06-10 | | 4 | Mary | Brown | HR | 55000.00 | 2022-02-01 | | 5 | David | Lee | IT | 80000.00 | 2018-09-01 |

After `SELECT WHERE Department = 'Sales'`: | FirstName | LastName | Salary | | :-------- | :------- | :------- | | John | Doe | 60000.00 | | Peter | Jones | 62000.00 |

After `UPDATE` and final `SELECT *`: | EmployeeID | FirstName | LastName | Department | Salary | HireDate | | :--------- | :-------- | :------- | :--------- | :------- | :--------- | | 1 | John | Doe | Sales | 60000.00 | 2020-01-15 | | 2 | Jane | Smith | Marketing | 75000.00 | 2019-03-20 | | 3 | Peter | Jones | IT | 62000.00 | 2021-06-10 | | 5 | David | Lee | IT | 80000.00 | 2018-09-01 |

Lab 4: Execution of join operations

Title: Execution of SQL Join Operations

Aim: To understand and demonstrate different types of SQL JOIN operations (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN) by combining data from multiple related tables.

Procedure:

1. **Connect to Database:** Establish a connection to your SQL database server.
2. **Create Tables:** Create two or more related tables (e.g., `Departments` and `Employees`) that have a common column to facilitate joining.
3. **Insert Data:** Populate both tables with sample data, ensuring some records have matching values in the common column and some do not, to observe the behavior of different join types.
4. **Perform INNER JOIN:** Execute an `INNER JOIN` query to retrieve only the rows that have matching values in both tables.
5. **Perform LEFT JOIN (or LEFT OUTER JOIN):** Execute a `LEFT JOIN` query to retrieve all rows from the left table and the matching rows from the right table. If there's no match, NULLs appear for the right table's columns.
6. **Perform RIGHT JOIN (or RIGHT OUTER JOIN):** Execute a `RIGHT JOIN` query to retrieve all rows from the right table and the matching rows from the left table. If there's no match, NULLs appear for the left table's columns.
7. **Perform FULL OUTER JOIN (if supported):** Execute a `FULL OUTER JOIN` query to retrieve all rows when there is a match in either the left or right table. If no match, NULLs appear for the non-matching side. (Note: Some databases like MySQL do not directly support `FULL OUTER JOIN` and it needs to be simulated using `UNION` of `LEFT JOIN` and `RIGHT JOIN`).
8. **Analyze Results:** Compare the output of each join type to understand their differences.

Source Code:

```
-- DDL: Create Departments table
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(100) NOT NULL
);

-- DDL: Create Employees table (with a foreign key to Departments)
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DepartmentID INT,
    Salary DECIMAL(10, 2),
    FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)
);

-- DML: Insert data into Departments
INSERT INTO Departments (DepartmentID, DepartmentName) VALUES
(1, 'Sales'),
(2, 'Marketing'),
(3, 'IT'),
(4, 'HR');

-- DML: Insert data into Employees
```

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID, Salary)
VALUES
(101, 'Alice', 'Smith', 1, 60000.00),
(102, 'Bob', 'Johnson', 2, 75000.00),
(103, 'Charlie', 'Brown', 1, 62000.00),
(104, 'Diana', 'Prince', 3, 80000.00),
(105, 'Eve', 'Davis', NULL, 50000.00); -- Employee with no department

-- SQL: INNER JOIN
-- Selects records that have matching values in both tables
SELECT E.FirstName, E.LastName, D.DepartmentName
FROM Employees AS E
INNER JOIN Departments AS D ON E.DepartmentID = D.DepartmentID;

-- SQL: LEFT JOIN (or LEFT OUTER JOIN)
-- Selects all records from the left table, and the matched records from the
right table
SELECT E.FirstName, E.LastName, D.DepartmentName
FROM Employees AS E
LEFT JOIN Departments AS D ON E.DepartmentID = D.DepartmentID;

-- SQL: RIGHT JOIN (or RIGHT OUTER JOIN)
-- Selects all records from the right table, and the matched records from the
left table
SELECT E.FirstName, E.LastName, D.DepartmentName
FROM Departments AS D
RIGHT JOIN Employees AS E ON D.DepartmentID = E.DepartmentID;

-- SQL: FULL OUTER JOIN (Conceptual - syntax varies by DB, e.g., SQL Server,
PostgreSQL)
-- For MySQL, simulate with UNION:
-- SELECT E.FirstName, E.LastName, D.DepartmentName
-- FROM Employees AS E
-- LEFT JOIN Departments AS D ON E.DepartmentID = D.DepartmentID
-- UNION
-- SELECT E.FirstName, E.LastName, D.DepartmentName
-- FROM Employees AS E
-- RIGHT JOIN Departments AS D ON E.DepartmentID = D.DepartmentID
-- WHERE E.DepartmentID IS NULL; -- To exclude duplicates from the left join
part

-- Example of FULL OUTER JOIN (if your DB supports it directly, e.g.,
PostgreSQL)
-- SELECT E.FirstName, E.LastName, D.DepartmentName
-- FROM Employees AS E
-- FULL OUTER JOIN Departments AS D ON E.DepartmentID = D.DepartmentID;

-- Optional: Drop tables (for cleanup)
-- DROP TABLE Employees;
-- DROP TABLE Departments;
```

Input: The SQL queries and the sample data inserted into `Departments` and `Employees` tables.

Expected Output:

Initial Tables:

Departments: | DepartmentID | DepartmentName | | :---------- | :---------- | | 1 | Sales | | 2 | Marketing | | 3 | IT | | 4 | HR |

Employees: | EmployeeID | FirstName | LastName | DepartmentID | Salary | | :--------- | :------- | :-------- | :---------- | :------- | | 101 | Alice | Smith | 1 | 60000.00 | | 102 | Bob | Johnson | 2 | 75000.00 | |

103 | Charlie | Brown | 1 | 62000.00 | | 104 | Diana | Prince | 3 | 80000.00 | | 105 | Eve | Davis | NULL | 50000.00 |

Output of INNER JOIN: | FirstName | LastName | DepartmentName | | :-------- | :------- | :---------- | | Alice | Smith | Sales | | Bob | Johnson | Marketing | | Charlie | Brown | Sales | | Diana | Prince | IT |

Output of LEFT JOIN: | FirstName | LastName | DepartmentName | | :-------- | :------- | :---------- | | Alice | Smith | Sales | | Bob | Johnson | Marketing | | Charlie | Brown | Sales | | Diana | Prince | IT | | Eve | Davis | NULL |

Output of RIGHT JOIN (with `Departments` as left and `Employees` as right): | FirstName | LastName | DepartmentName | | :-------- | :------- | :---------- | | Alice | Smith | Sales | | Bob | Johnson | Marketing | | Charlie | Brown | Sales | | Diana | Prince | IT | | Eve | Davis | NULL | | NULL | NULL | HR |

(Note: The `RIGHT JOIN` here is on `Departments AS D RIGHT JOIN Employees AS E`, meaning all `Employees` records are kept, and matching `Departments` are included. If `Departments` had a record with no matching `Employee`, that would also appear with `NULL` for employee columns.)

Output of FULL OUTER JOIN (Conceptual, if supported): | FirstName | LastName | DepartmentName | | :-------- | :------- | :---------- | | Alice | Smith | Sales | | Bob | Johnson | Marketing | | Charlie | Brown | Sales | | Diana | Prince | IT | | Eve | Davis | NULL | | NULL | NULL | HR |

Lab 5: Practice of triggers-SQL Trigger | Student Database

Title: SQL Triggers for Student Database

Aim: To learn and implement SQL triggers to automate actions on a student database in response to specific data modification events (INSERT, UPDATE, DELETE), such as logging changes or enforcing business rules.

Procedure:

1. **Connect to Database:** Establish a connection to your SQL database server.
2. **Create Student Table:** Create a `Students` table (if not already existing from Lab 2).
3. **Create Audit Table:** Create a separate `StudentAuditLog` table to record changes made to the `Students` table. This table will typically include columns like `LogID`, `StudentID`, `ChangeType`, `OldValue`, `NewValue`, `ChangeDate`, `ChangedBy`.
4. **Define Trigger:** Write a `CREATE TRIGGER` statement. Specify:
    - **Trigger Name:** A descriptive name (e.g., `trg_StudentChanges`).
    - **Timing:** `BEFORE` or `AFTER` the event.
    - **Event:** `INSERT`, `UPDATE`, or `DELETE` (or a combination).
    - **Table:** The table on which the trigger will fire (`Students`).
    - **Action:** The SQL statements to execute when the trigger fires (e.g., `INSERT` into `StudentAuditLog`).
    - Use `OLD` and `NEW` pseudo-tables (syntax varies by database, e.g., `OLD.column_name`, `NEW.column_name`) to access data before and after the modification.
5. **Test Trigger:** Perform `INSERT`, `UPDATE`, and `DELETE` operations on the `Students` table.
6. **Verify Audit Log:** Query the `StudentAuditLog` table to confirm that the trigger fired correctly and recorded the changes.

Source Code:

```
-- DDL: Create Students table (if not exists)
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Major VARCHAR(100),
    EnrollmentDate DATE
);

-- DDL: Create an audit log table
CREATE TABLE StudentAuditLog (
    LogID INT PRIMARY KEY AUTO_INCREMENT, -- AUTO_INCREMENT for MySQL, IDENTITY
for SQL Server, SERIAL for PostgreSQL
    StudentID INT,
    ChangeType VARCHAR(10), -- e.g., 'INSERT', 'UPDATE', 'DELETE'
    OldMajor VARCHAR(100),
    NewMajor VARCHAR(100),
    ChangeDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    ChangedBy VARCHAR(100) -- In a real system, this would be a user ID
);

-- DML: Insert some initial data
INSERT INTO Students (StudentID, FirstName, LastName, Major, EnrollmentDate)
VALUES
(101, 'Alice', 'Smith', 'Computer Science', '2022-09-01'),
(102, 'Bob', 'Johnson', 'Electrical Engineering', '2022-09-01');
```

```
-- SQL: Create an AFTER UPDATE trigger on Students table
-- This trigger logs changes to the 'Major' column
DELIMITER // -- For MySQL, to change delimiter for trigger creation
CREATE TRIGGER trg_StudentMajorChanges
AFTER UPDATE ON Students
FOR EACH ROW
BEGIN
    IF OLD.Major <> NEW.Major THEN
        INSERT INTO StudentAuditLog (StudentID, ChangeType, OldMajor, NewMajor,
ChangedBy)
        VALUES (OLD.StudentID, 'UPDATE', OLD.Major, NEW.Major, USER()); --
USER() gets current database user
    END IF;
END;
//
DELIMITER ; -- Reset delimiter for MySQL

-- SQL: Create an AFTER INSERT trigger on Students table
-- This trigger logs new student enrollments
DELIMITER //
CREATE TRIGGER trg_StudentNewEnrollment
AFTER INSERT ON Students
FOR EACH ROW
BEGIN
    INSERT INTO StudentAuditLog (StudentID, ChangeType, NewMajor, ChangedBy)
    VALUES (NEW.StudentID, 'INSERT', NEW.Major, USER());
END;
//
DELIMITER ;

-- SQL: Create an AFTER DELETE trigger on Students table
-- This trigger logs student deletions
DELIMITER //
CREATE TRIGGER trg_StudentDeletion
AFTER DELETE ON Students
FOR EACH ROW
BEGIN
    INSERT INTO StudentAuditLog (StudentID, ChangeType, OldMajor, ChangedBy)
    VALUES (OLD.StudentID, 'DELETE', OLD.Major, USER());
END;
//
DELIMITER ;

-- DML: Test the triggers
-- 1. Insert a new student (should trigger trg_StudentNewEnrollment)
INSERT INTO Students (StudentID, FirstName, LastName, Major, EnrollmentDate)
VALUES
(103, 'Charlie', 'Brown', 'Mathematics', '2023-01-15');

-- 2. Update a student's major (should trigger trg_StudentMajorChanges)
UPDATE Students SET Major = 'Software Engineering' WHERE StudentID = 101;

-- 3. Delete a student (should trigger trg_StudentDeletion)
DELETE FROM Students WHERE StudentID = 102;

-- SQL: Verify the audit log
SELECT * FROM StudentAuditLog;

-- Optional: Drop triggers and tables (for cleanup)
-- DROP TRIGGER IF EXISTS trg_StudentMajorChanges;
-- DROP TRIGGER IF EXISTS trg_StudentNewEnrollment;
-- DROP TRIGGER IF EXISTS trg_StudentDeletion;
-- DROP TABLE IF EXISTS StudentAuditLog;
-- DROP TABLE IF EXISTS Students;
```

Input: The SQL DDL and DML statements, specifically the `INSERT`, `UPDATE`, and `DELETE` commands that activate the triggers.

Expected Output:

Content of `StudentAuditLog` after all test operations: (Note: `LogID` and `ChangeDate` will vary)

LogID	StudentID	ChangeType	OldMajor	NewMajor	ChangeDate	ChangedBy
1	103	INSERT	NULL	Mathematics	(timestamp)	(user)
2	101	UPDATE	Computer Science	Software Engineering	(timestamp)	(user)
3	102	DELETE	Electrical Engineering	NULL	(timestamp)	(user)

Lab 6: Practice of triggers-SQL Trigger | Employee Database

Title: SQL Triggers for Employee Database

Aim: To apply SQL triggers to an employee database to enforce business rules, maintain data integrity, or create audit trails in response to data modifications.

Procedure:

1. **Connect to Database:** Establish a connection to your SQL database server.
2. **Create Employee Table:** Create an `Employees` table (if not already existing from Lab 3).
3. **Create Audit Table:** Create an `EmployeeAuditLog` table to record changes to employee data, similar to the student audit log.
4. **Define Trigger(s):** Write `CREATE TRIGGER` statements for the `Employees` table. Examples:
    - A trigger to prevent salary updates that are less than the current salary.
    - A trigger to log all `INSERT`, `UPDATE`, `DELETE` operations on employee records.
    - A trigger to automatically update a `LastModifiedDate` column on `UPDATE`.
5. **Test Trigger(s):** Perform various `INSERT`, `UPDATE`, and `DELETE` operations on the `Employees` table to test the defined triggers.
6. **Verify Results:** Query the `EmployeeAuditLog` or check the modified `Employees` table to confirm trigger functionality.

Source Code:

```
-- DDL: Create Employees table (if not exists)
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Department VARCHAR(100),
    Salary DECIMAL(10, 2),
    HireDate DATE,
    LastModifiedDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP -- Auto-updates on modification
);

-- DDL: Create an audit log table for employees
CREATE TABLE EmployeeAuditLog (
    LogID INT PRIMARY KEY AUTO_INCREMENT,
    EmployeeID INT,
    ChangeType VARCHAR(10),
    Description VARCHAR(255),
    ChangeDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    ChangedBy VARCHAR(100)
);

-- DML: Insert some initial data
INSERT INTO Employees (EmployeeID, FirstName, LastName, Department, Salary,
HireDate) VALUES
(101, 'Alice', 'Smith', 'Sales', 60000.00, '2020-01-15'),
(102, 'Bob', 'Johnson', 'Marketing', 75000.00, '2019-03-20');

-- SQL: Create an AFTER INSERT trigger to log new employees
DELIMITER //
CREATE TRIGGER trg_EmployeeInsertLog
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
```

```
    INSERT INTO EmployeeAuditLog (EmployeeID, ChangeType, Description,
ChangedBy)
    VALUES (NEW.EmployeeID, 'INSERT', CONCAT('New employee ', NEW.FirstName, '
', NEW.LastName, ' added.'), USER());
END;
//
DELIMITER ;

-- SQL: Create a BEFORE UPDATE trigger to prevent salary decrease
-- This trigger will raise an error if salary is attempted to be decreased
DELIMITER //
CREATE TRIGGER trg_PreventSalaryDecrease
BEFORE UPDATE ON Employees
FOR EACH ROW
BEGIN
    IF NEW.Salary < OLD.Salary THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Salary cannot be
decreased!';
    END IF;
END;
//
DELIMITER ;

-- SQL: Create an AFTER UPDATE trigger to log salary changes
DELIMITER //
CREATE TRIGGER trg_EmployeeSalaryUpdateLog
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
    IF OLD.Salary <> NEW.Salary THEN
        INSERT INTO EmployeeAuditLog (EmployeeID, ChangeType, Description,
ChangedBy)
        VALUES (OLD.EmployeeID, 'UPDATE', CONCAT('Salary changed from ',
OLD.Salary, ' to ', NEW.Salary), USER());
    END IF;
END;
//
DELIMITER ;

-- DML: Test the triggers
-- 1. Insert a new employee (should trigger trg_EmployeeInsertLog)
INSERT INTO Employees (EmployeeID, FirstName, LastName, Department, Salary,
HireDate) VALUES
(103, 'Charlie', 'Brown', 'IT', 80000.00, '2023-05-10');

-- 2. Update employee salary (should trigger trg_EmployeeSalaryUpdateLog)
UPDATE Employees SET Salary = 65000.00 WHERE EmployeeID = 101;

-- 3. Attempt to decrease salary (should trigger trg_PreventSalaryDecrease and
raise an error)
-- This statement is expected to fail. Uncomment to test.
-- UPDATE Employees SET Salary = 50000.00 WHERE EmployeeID = 101;

-- 4. Update employee department (LastModifiedDate should update automatically)
UPDATE Employees SET Department = 'Marketing' WHERE EmployeeID = 101;

-- SQL: Verify the audit log
SELECT * FROM EmployeeAuditLog;

-- SQL: Verify the Employees table (especially LastModifiedDate)
SELECT * FROM Employees;

-- Optional: Drop triggers and tables (for cleanup)
-- DROP TRIGGER IF EXISTS trg_EmployeeInsertLog;
-- DROP TRIGGER IF EXISTS trg_PreventSalaryDecrease;
-- DROP TRIGGER IF EXISTS trg_EmployeeSalaryUpdateLog;
-- DROP TABLE IF EXISTS EmployeeAuditLog;
```

```
-- DROP TABLE IF EXISTS Employees;
```

Input: The SQL DDL and DML statements, including the `INSERT` and `UPDATE` commands used to test the triggers.

Expected Output:

Output of `SELECT * FROM EmployeeAuditLog;` (after successful operations): (Note: `LogID` and `ChangeDate` will vary)

LogID	EmployeeID	ChangeType	Description	ChangeDate	ChangedBy
1	103	INSERT	New employee Charlie Brown added.	(timestamp)	(user)
2	101	UPDATE	Salary changed from 60000.00 to 65000.00	(timestamp)	(user)

Output of `SELECT * FROM Employees;` (after successful operations):

EmployeeID	FirstName	LastName	Department	Salary	HireDate	LastModifiedDate
101	Alice	Smith	Marketing	65000.00	2020-01-15	(latest timestamp)
102	Bob	Johnson	Marketing	75000.00	2019-03-20	(initial timestamp)
103	Charlie	Brown	IT	80000.00	2023-05-10	(initial timestamp)

If `UPDATE Employees SET Salary = 50000.00 WHERE EmployeeID = 101;` is executed: An error message similar to: `ERROR 1644 (45000): Salary cannot be decreased!`

Lab 7: Sample programs for cursors

Title: SQL Cursors

Aim: To understand the concept of SQL cursors and implement them for row-by-row processing of a result set, demonstrating scenarios where cursors might be necessary (e.g., complex calculations per row, calling external procedures).

Procedure:

1. **Connect to Database:** Establish a connection to your SQL database server.
2. **Create Sample Table:** Create a table with sample data that will be processed by the cursor.
3. **Declare Cursor:** Declare a cursor, associating it with a `SELECT` statement that defines the result set to be traversed.
4. **Open Cursor:** Open the cursor to populate it with the result set.
5. **Fetch Data:** Use a loop to fetch rows one by one from the cursor into variables.
6. **Process Data:** Perform operations on the fetched data within the loop (e.g., print values, perform calculations, update another table).
7. **Close Cursor:** Close the cursor after processing all rows.
8. **Deallocate Cursor:** Deallocate the cursor to release database resources.

Source Code:

(Note: Cursor syntax varies significantly between SQL dialects. Below is an example for MySQL using stored procedures. Similar concepts apply to PL/SQL for Oracle or T-SQL for SQL Server.)

```
-- DDL: Create a sample Products table
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100) NOT NULL,
    Price DECIMAL(10, 2),
    StockQuantity INT
);

-- DML: Insert sample data
INSERT INTO Products (ProductID, ProductName, Price, StockQuantity) VALUES
(1, 'Laptop', 1200.00, 50),
(2, 'Mouse', 25.00, 200),
(3, 'Keyboard', 75.00, 150),
(4, 'Monitor', 300.00, 80);

-- SQL: Example Stored Procedure using a Cursor to apply a discount
DELIMITER //

CREATE PROCEDURE ApplyDiscountToLowStockProducts(IN discount_percentage
DECIMAL(5,2))
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE p_id INT;
    DECLARE p_name VARCHAR(100);
    DECLARE p_price DECIMAL(10, 2);
    DECLARE p_stock INT;

    -- Declare cursor for products with stock quantity less than 100
    DECLARE product_cursor CURSOR FOR
        SELECT ProductID, ProductName, Price, StockQuantity
        FROM Products
```

```
        WHERE StockQuantity < 100;

    -- Declare continue handler for not found (end of cursor)
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    -- Open the cursor
    OPEN product_cursor;

    -- Loop through the result set
    read_loop: LOOP
        FETCH product_cursor INTO p_id, p_name, p_price, p_stock;

        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Apply discount and update the product
        UPDATE Products
        SET Price = p_price * (1 - discount_percentage / 100)
        WHERE ProductID = p_id;

        SELECT CONCAT('Applied discount to ', p_name, '. New price: ', p_price *
(1 - discount_percentage / 100)) AS Status;
    END LOOP;

    -- Close the cursor
    CLOSE product_cursor;

END //

DELIMITER ;

-- DML: Call the stored procedure to apply a 10% discount
CALL ApplyDiscountToLowStockProducts(10.00);

-- SQL: Verify the updated prices
SELECT * FROM Products;

-- Optional: Drop procedure and table (for cleanup)
-- DROP PROCEDURE IF EXISTS ApplyDiscountToLowStockProducts;
-- DROP TABLE IF EXISTS Products;
```

Input: The SQL DDL, DML, and the call to the stored procedure `ApplyDiscountToLowStockProducts` with a discount percentage.

Expected Output:

Output from `CALL ApplyDiscountToLowStockProducts(10.00);`: (Each row represents a `SELECT CONCAT(...)` from inside the loop)

Status
Applied discount to Laptop. New price: 1080.00
Applied discount to Monitor. New price: 270.00

Output of `SELECT * FROM Products;` after procedure execution:

ProductID	ProductName	Price	StockQuantity
1	Laptop	1080.00	50
2	Mouse	25.00	200

3	Keyboard	75.00	150
4	Monitor	270.00	80

Lab 8: Case study for JDBC

Title: JDBC Case Study

Aim: To understand and apply Java Database Connectivity (JDBC) to connect a Java application to a relational database, perform basic CRUD (Create, Read, Update, Delete) operations, and handle exceptions.

Procedure:

1. **Setup Development Environment:** Ensure Java Development Kit (JDK) is installed. Set up an IDE (e.g., Eclipse, IntelliJ IDEA, VS Code).
2. **Obtain JDBC Driver:** Download the appropriate JDBC driver (JAR file) for your specific database (e.g., MySQL Connector/J, PostgreSQL JDBC Driver). Add this JAR to your project's classpath.
3. **Create Database and Table:** Prepare a simple database and a table (e.g., `Books` table with `BookID`, `Title`, `Author`, `Price`) for the Java application to interact with.
4. **Write Java Code:**
    - Import necessary JDBC classes (`java.sql.*`).
    - Load the JDBC driver class.
    - Establish a connection to the database using `DriverManager.getConnection()`.
    - Create `Statement` or `PreparedStatement` objects to execute SQL queries.
    - Execute DML (`INSERT`, `UPDATE`, `DELETE`) and DQL (`SELECT`) queries.
    - Process `ResultSet` for `SELECT` queries.
    - Handle `SQLException` using `try-catch` blocks.
    - Close all JDBC resources (`ResultSet`, `Statement`, `Connection`) in a `finally` block to prevent resource leaks.
5. **Compile and Run:** Compile the Java code and execute it.
6. **Verify Database:** Check the database directly using a SQL client to confirm that changes made by the Java application are reflected.

Source Code:

```
// Save this as 'JdbcBookManager.java'
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcBookManager {

    // Database credentials and URL (replace with your actual details)
    static final String DB_URL = "jdbc:mysql://localhost:3306/mydatabase"; // Or
jdbc:postgresql://localhost:5432/mydatabase
    static final String USER = "your_username";
    static final String PASS = "your_password";

    public static void main(String[] args) {
        Connection conn = null;
        Statement stmt = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
```

```
            // STEP 1: Register JDBC driver (for older JDBC versions, often not
needed for modern drivers)
            // Class.forName("com.mysql.cj.jdbc.Driver"); // For MySQL 8+
            // Class.forName("org.postgresql.Driver"); // For PostgreSQL

            System.out.println("Connecting to database...");
            // STEP 2: Open a connection
            conn = DriverManager.getConnection(DB_URL, USER, PASS);
            System.out.println("Connected successfully!");

            // STEP 3: Create a table (if it doesn't exist)
            stmt = conn.createStatement();
            String createTableSQL = "CREATE TABLE IF NOT EXISTS Books (" +
                                    "BookID INT PRIMARY KEY AUTO_INCREMENT," +
// Use SERIAL for PostgreSQL
                                    "Title VARCHAR(255) NOT NULL," +
                                    "Author VARCHAR(100) NOT NULL," +
                                    "Price DECIMAL(10, 2)" +
                                    ")";
            stmt.executeUpdate(createTableSQL);
            System.out.println("Books table created or already exists.");

            // STEP 4: Insert a new book (CREATE)
            System.out.println("\nInserting a new book...");
            String insertSQL = "INSERT INTO Books (Title, Author, Price) VALUES
(?, ?, ?)";
            pstmt = conn.prepareStatement(insertSQL);
            pstmt.setString(1, "The Great Gatsby");
            pstmt.setString(2, "F. Scott Fitzgerald");
            pstmt.setDouble(3, 15.99);
            int rowsAffected = pstmt.executeUpdate();
            System.out.println(rowsAffected + " row(s) inserted.");

            // STEP 5: Read all books (READ)
            System.out.println("\nRetrieving all books:");
            String selectSQL = "SELECT BookID, Title, Author, Price FROM Books";
            rs = stmt.executeQuery(selectSQL);

            while (rs.next()) {
                int id = rs.getInt("BookID");
                String title = rs.getString("Title");
                String author = rs.getString("Author");
                double price = rs.getDouble("Price");
                System.out.printf("ID: %d, Title: %s, Author: %s, Price:
%.2f%n", id, title, author, price);
            }

            // STEP 6: Update a book's price (UPDATE)
            System.out.println("\nUpdating book price...");
            String updateSQL = "UPDATE Books SET Price = ? WHERE Title = ?";
            pstmt = conn.prepareStatement(updateSQL);
            pstmt.setDouble(1, 18.50);
            pstmt.setString(2, "The Great Gatsby");
            rowsAffected = pstmt.executeUpdate();
            System.out.println(rowsAffected + " row(s) updated.");

            // Verify update
            System.out.println("\nVerifying updated book:");
            String selectUpdatedSQL = "SELECT BookID, Title, Author, Price FROM
Books WHERE Title = 'The Great Gatsby'";
            rs = stmt.executeQuery(selectUpdatedSQL);
            if (rs.next()) {
                System.out.printf("ID: %d, Title: %s, Author: %s, Price:
%.2f%n",
                                  rs.getInt("BookID"), rs.getString("Title"),
                                  rs.getString("Author"),
rs.getDouble("Price"));
```

```
            }

            // STEP 7: Delete a book (DELETE)
            System.out.println("\nDeleting a book...");
            String deleteSQL = "DELETE FROM Books WHERE Title = ?";
            pstmt = conn.prepareStatement(deleteSQL);
            pstmt.setString(1, "The Great Gatsby");
            rowsAffected = pstmt.executeUpdate();
            System.out.println(rowsAffected + " row(s) deleted.");

            // Verify deletion
            System.out.println("\nVerifying deletion (should be empty):");
            rs = stmt.executeQuery(selectSQL);
            if (!rs.next()) {
                System.out.println("No books found.");
            }

        } catch (SQLException se) {
            // Handle errors for JDBC
            se.printStackTrace();
        } catch (Exception e) {
            // Handle errors for Class.forName
            e.printStackTrace();
        } finally {
            // STEP 8: Close resources
            try {
                if (rs != null) rs.close();
            } catch (SQLException se2) {
                // Nothing to do
            }
            try {
                if (pstmt != null) pstmt.close();
            } catch (SQLException se2) {
                // Nothing to do
            }
            try {
                if (stmt != null) stmt.close();
            } catch (SQLException se2) {
                // Nothing to do
            }
            try {
                if (conn != null) conn.close();
            } catch (SQLException se) {
                se.printStackTrace();
            }
            System.out.println("\nGoodbye!");
        }
    }
}
```

Input:

- A running database server (e.g., MySQL or PostgreSQL).
- The JDBC driver JAR file placed in the project's classpath.
- The Java source code (`JdbcBookManager.java`).
- Database connection details (URL, username, password) configured in the Java code.

Expected Output:

```
Connecting to database...
Connected successfully!
Books table created or already exists.

Inserting a new book...
```

```
1 row(s) inserted.

Retrieving all books:
ID: 1, Title: The Great Gatsby, Author: F. Scott Fitzgerald, Price: 15.99

Updating book price...
1 row(s) updated.

Verifying updated book:
ID: 1, Title: The Great Gatsby, Author: F. Scott Fitzgerald, Price: 18.50

Deleting a book...
1 row(s) deleted.

Verifying deletion (should be empty):
No books found.

Goodbye!
```

(Note: Actual `BookID` will depend on auto-increment sequence. If the table already exists and has data, the initial `SELECT` will show existing data plus the newly inserted book.)

Lab 9: Creating a student database

Title: Creating a Student Database Schema

Aim: To design and implement a relational database schema for managing student information, including defining tables, columns, data types, primary keys, foreign keys, and other constraints.

Procedure:

1. **Understand Requirements:** Identify the core entities in a student management system (e.g., Students, Courses, Enrollments, Instructors, Departments).
2. **Design Tables:** For each entity, determine the necessary attributes and map them to appropriate SQL data types.
3. **Define Primary Keys:** Choose a unique identifier (primary key) for each table.
4. **Establish Relationships:** Identify relationships between tables and define foreign keys to enforce referential integrity. Specify `ON DELETE` and `ON UPDATE` actions (e.g., `CASCADE`, `RESTRICT`, `SET NULL`).
5. **Add Constraints:** Implement other constraints like `NOT NULL`, `UNIQUE`, `DEFAULT`, and `CHECK` constraints to ensure data quality.
6. **Write DDL Statements:** Translate the design into SQL `CREATE TABLE` statements.
7. **Execute DDL:** Execute the SQL statements in your database management system.
8. **Verify Schema:** Use database tools to inspect the created tables, their columns, and constraints.

Source Code:

```
-- DDL: Create the Student Management Database (if your DBMS supports it)
-- CREATE DATABASE StudentManagementDB;
-- USE StudentManagementDB; -- For MySQL

-- DDL: Create the Departments Table
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY AUTO_INCREMENT, -- SERIAL for PostgreSQL,
IDENTITY for SQL Server
    DepartmentName VARCHAR(100) NOT NULL UNIQUE
);

-- DDL: Create the Students Table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY AUTO_INCREMENT,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DateOfBirth DATE,
    Email VARCHAR(100) UNIQUE NOT NULL,
    PhoneNumber VARCHAR(20),
    EnrollmentDate DATE DEFAULT CURRENT_DATE,
    DepartmentID INT,
    CONSTRAINT FK_Department FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID)
        ON DELETE SET NULL -- If a department is deleted, students in it will
have NULL DepartmentID
        ON UPDATE CASCADE   -- If a department ID changes, update in Students
table
);

-- DDL: Create the Courses Table
CREATE TABLE Courses (
```

```
    CourseID VARCHAR(10) PRIMARY KEY, -- e.g., 'CS101', 'EE205'
    CourseName VARCHAR(100) NOT NULL,
    Credits INT NOT NULL CHECK (Credits > 0 AND Credits <= 6),
    DepartmentID INT,
    CONSTRAINT FK_CourseDepartment FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID)
        ON DELETE CASCADE -- If a department is deleted, its courses are also
deleted
        ON UPDATE CASCADE
);

-- DDL: Create the Instructors Table
CREATE TABLE Instructors (
    InstructorID INT PRIMARY KEY AUTO_INCREMENT,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE NOT NULL,
    DepartmentID INT,
    CONSTRAINT FK_InstructorDepartment FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);

-- DDL: Create the Enrollments Table (Junction table for Many-to-Many
relationship between Students and Courses)
CREATE TABLE Enrollments (
    EnrollmentID INT PRIMARY KEY AUTO_INCREMENT,
    StudentID INT NOT NULL,
    CourseID VARCHAR(10) NOT NULL,
    EnrollmentDate DATE DEFAULT CURRENT_DATE,
    Grade CHAR(2), -- e.g., 'A+', 'B', 'C-'
    CONSTRAINT FK_EnrollmentStudent FOREIGN KEY (StudentID) REFERENCES
Students(StudentID)
        ON DELETE CASCADE -- If a student is deleted, their enrollments are also
deleted
        ON UPDATE CASCADE,
    CONSTRAINT FK_EnrollmentCourse FOREIGN KEY (CourseID) REFERENCES
Courses(CourseID)
        ON DELETE CASCADE -- If a course is deleted, its enrollments are also
deleted
        ON UPDATE CASCADE,
    CONSTRAINT UQ_StudentCourse UNIQUE (StudentID, CourseID) -- A student can
enroll in a course only once
);

-- DDL: Create a table for Course Offerings (linking Courses and Instructors)
CREATE TABLE CourseOfferings (
    OfferingID INT PRIMARY KEY AUTO_INCREMENT,
    CourseID VARCHAR(10) NOT NULL,
    InstructorID INT NOT NULL,
    AcademicYear INT NOT NULL,
    Semester VARCHAR(10) NOT NULL, -- e.g., 'Fall', 'Spring', 'Summer'
    CONSTRAINT FK_OfferingCourse FOREIGN KEY (CourseID) REFERENCES
Courses(CourseID)
        ON DELETE CASCADE
        ON UPDATE CASCADE,
    CONSTRAINT FK_OfferingInstructor FOREIGN KEY (InstructorID) REFERENCES
Instructors(InstructorID)
        ON DELETE RESTRICT, -- Prevent deleting an instructor if they are
currently teaching a course
    CONSTRAINT UQ_CourseInstructorYearSemester UNIQUE (CourseID, InstructorID,
AcademicYear, Semester)
);

-- DML: Sample Data Insertion (Optional, but good for testing schema)
```

```
INSERT INTO Departments (DepartmentName) VALUES ('Computer Science'),
('Electrical Engineering'), ('Mathematics');

INSERT INTO Students (FirstName, LastName, DateOfBirth, Email, DepartmentID)
VALUES
('Alice', 'Smith', '2003-05-15', 'alice.s@university.edu', 1),
('Bob', 'Johnson', '2002-11-22', 'bob.j@university.edu', 2),
('Charlie', 'Brown', '2004-01-30', 'charlie.b@university.edu', 1);

INSERT INTO Courses (CourseID, CourseName, Credits, DepartmentID) VALUES
('CS101', 'Introduction to Programming', 3, 1),
('EE205', 'Circuit Analysis', 4, 2),
('MA101', 'Calculus I', 3, 3);

INSERT INTO Instructors (FirstName, LastName, Email, DepartmentID) VALUES
('Dr. Emily', 'White', 'emily.w@university.edu', 1),
('Prof. Robert', 'Green', 'robert.g@university.edu', 2);

INSERT INTO Enrollments (StudentID, CourseID, Grade) VALUES
(1, 'CS101', 'A'),
(2, 'EE205', 'B+'),
(3, 'CS101', 'B');

INSERT INTO CourseOfferings (CourseID, InstructorID, AcademicYear, Semester)
VALUES
('CS101', 1, 2024, 'Fall'),
('EE205', 2, 2024, 'Fall');
```

Input: The SQL DDL statements provided in the source code.

Expected Output: A successfully created relational database schema named `StudentManagementDB` (or similar, depending on DBMS) containing the following tables with their defined columns, data types, primary keys, foreign keys, and other constraints:

- `Departments`
- `Students`
- `Courses`
- `Instructors`
- `Enrollments`
- `CourseOfferings`

You can verify this by using database-specific commands like `DESCRIBE TableName;` (MySQL), `\d TableName;` (PostgreSQL), or `sp_help TableName;` (SQL Server).

Lab 10: Create an XML document for employee information

Title: Creating an XML Document for Employee Information

Aim: To understand the structure and syntax of XML (Extensible Markup Language) and create a well-formed XML document to represent hierarchical employee data.

Procedure:

1. **Understand XML Structure:** Familiarize yourself with XML elements, attributes, root element, child elements, and the rules for well-formed XML (e.g., single root, matching tags, proper nesting).
2. **Define Root Element:** Choose a suitable root element for the entire document (e.g., `<Employees>`).
3. **Define Employee Element:** Create a repeating element for each employee (e.g., `<Employee>`).
4. **Define Employee Attributes/Elements:** Decide which employee properties will be represented as attributes (e.g., `id` for `<Employee>`) and which as nested elements (e.g., `<FirstName>`, `<LastName>`, `<Department>`, `<Salary>`).
5. **Add Sample Data:** Populate the XML structure with realistic employee data.
6. **Save as .xml:** Save the content in a file with a `.xml` extension.
7. **Validate (Optional):** If a DTD (Document Type Definition) or XML Schema is provided, validate the XML document against it to ensure it is valid.

Source Code:

```
<?xml version="1.0" encoding="UTF-8"?>
<Employees>
    <Employee id="EMP001">
        <PersonalDetails>
            <FirstName>John</FirstName>
            <LastName>Doe</LastName>
            <DateOfBirth>1985-03-20</DateOfBirth>
        </PersonalDetails>
        <ContactInfo>
            <Email>john.doe@example.com</Email>
            <Phone>+1-555-123-4567</Phone>
        </ContactInfo>
        <EmploymentDetails>
            <Department>Sales</Department>
            <Position>Sales Manager</Position>
            <Salary currency="USD">75000.00</Salary>
            <HireDate>2018-07-01</HireDate>
            <Status>Active</Status>
        </EmploymentDetails>
    </Employee>

    <Employee id="EMP002">
        <PersonalDetails>
            <FirstName>Jane</FirstName>
            <LastName>Smith</LastName>
            <DateOfBirth>1990-11-10</DateOfBirth>
        </PersonalDetails>
        <ContactInfo>
            <Email>jane.smith@example.com</Email>
            <Phone>+1-555-987-6543</Phone>
        </ContactInfo>
```

```
        <EmploymentDetails>
            <Department>Marketing</Department>
            <Position>Marketing Specialist</Position>
            <Salary currency="USD">60000.00</Salary>
            <HireDate>2020-01-20</HireDate>
            <Status>Active</Status>
        </EmploymentDetails>
    </Employee>

    <Employee id="EMP003">
        <PersonalDetails>
            <FirstName>Peter</FirstName>
            <LastName>Jones</LastName>
            <DateOfBirth>1992-06-25</DateOfBirth>
        </PersonalDetails>
        <ContactInfo>
            <Email>peter.j@example.com</Email>
            <Phone>+1-555-111-2222</Phone>
        </ContactInfo>
        <EmploymentDetails>
            <Department>IT</Department>
            <Position>Software Engineer</Position>
            <Salary currency="USD">90000.00</Salary>
            <HireDate>2019-09-01</HireDate>
            <Status>Active</Status>
        </EmploymentDetails>
    </Employee>

</Employees>
```

Input: Conceptual employee data that needs to be structured in an XML format.

Expected Output: A well-formed XML document (`employee_info.xml` or similar) containing the hierarchical representation of employee information, adhering to XML syntax rules. The document should be viewable in any web browser or XML editor.

Lab 11: Simple program for joins

Title: Simple Program for SQL Joins

Aim: To reinforce the understanding of SQL JOIN operations through a concise, practical example, focusing on how to combine data from two tables based on a common field.

Procedure:

1. **Connect to Database:** Establish a connection to your SQL database server.
2. **Create Two Tables:** Create two simple tables that have a logical relationship. For example, a `Customers` table and an `Orders` table, where `Orders` has a `CustomerID` foreign key referencing `Customers`.
3. **Insert Sample Data:** Populate both tables with a small set of data, ensuring some customers have orders and some do not, and some orders might not have a matching customer (though this would indicate data inconsistency in a real scenario).
4. **Execute INNER JOIN:** Write and execute an `INNER JOIN` query to retrieve customers who have placed orders, along with their order details.
5. **Execute LEFT JOIN:** Write and execute a `LEFT JOIN` query to retrieve all customers, and their order details if they have any.
6. **Analyze Results:** Observe the differences in the result sets produced by the `INNER JOIN` and `LEFT JOIN`.

Source Code:

```
-- DDL: Create Customers table
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    CustomerName VARCHAR(100) NOT NULL,
    City VARCHAR(50)
);

-- DDL: Create Orders table
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    Amount DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

-- DML: Insert data into Customers
INSERT INTO Customers (CustomerID, CustomerName, City) VALUES
(1, 'Alice Wonderland', 'New York'),
(2, 'Bob The Builder', 'London'),
(3, 'Charlie Chaplin', 'Paris');

-- DML: Insert data into Orders
INSERT INTO Orders (OrderID, CustomerID, OrderDate, Amount) VALUES
(101, 1, '2023-01-10', 150.00),
(102, 1, '2023-01-12', 200.50),
(103, 2, '2023-01-15', 75.25),
(104, 99, '2023-01-18', 300.00); -- Order with non-existent CustomerID (for
demonstrating joins)

-- SQL: INNER JOIN - Get customers who have placed orders
SELECT C.CustomerName, O.OrderID, O.OrderDate, O.Amount
```

```
FROM Customers AS C
INNER JOIN Orders AS O ON C.CustomerID = O.CustomerID;

-- SQL: LEFT JOIN - Get all customers and their orders (if any)
SELECT C.CustomerName, O.OrderID, O.OrderDate, O.Amount
FROM Customers AS C
LEFT JOIN Orders AS O ON C.CustomerID = O.CustomerID;

-- Optional: Drop tables (for cleanup)
-- DROP TABLE Orders;
-- DROP TABLE Customers;
```

Input: The SQL queries and the sample data inserted into the `Customers` and `Orders` tables.

Expected Output:

Initial Tables:

Customers: | CustomerID | CustomerName | City | | :--------- | :------------- | :--------- | | 1 | Alice Wonderland | New York | | 2 | Bob The Builder | London | | 3 | Charlie Chaplin | Paris |

Orders: | OrderID | CustomerID | OrderDate | Amount | | :------ | :--------- | :--------- | :------ | | 101 | 1 | 2023-01-10 | 150.00 | | 102 | 1 | 2023-01-12 | 200.50 | | 103 | 2 | 2023-01-15 | 75.25 | | 104 | 99 | 2023-01-18 | 300.00 |

Output of INNER JOIN: | CustomerName | OrderID | OrderDate | Amount | | :------------- | :------ | :--------- | :------ | | Alice Wonderland | 101 | 2023-01-10 | 150.00 | | Alice Wonderland | 102 | 2023-01-12 | 200.50 | | Bob The Builder | 103 | 2023-01-15 | 75.25 |

Output of LEFT JOIN: | CustomerName | OrderID | OrderDate | Amount | | :------------- | :------ | :--------- | :------ | | Alice Wonderland | 101 | 2023-01-10 | 150.00 | | Alice Wonderland | 102 | 2023-01-12 | 200.50 | | Bob The Builder | 103 | 2023-01-15 | 75.25 | | Charlie Chaplin | NULL | NULL | NULL |

Lab 12: Study of normalization techniques

Title: Study of Normalization Techniques

Aim: To understand the principles of database normalization (1NF, 2NF, 3NF, BCNF) and apply them to reduce data redundancy, eliminate data anomalies (insertion, update, deletion), and improve data integrity and consistency.

Procedure:

1. **Understand Unnormalized Form (UNF):** Start with a table that contains repeating groups or multiple values in a single column.
2. **Apply 1NF (First Normal Form):**
    - Eliminate repeating groups by placing each repeating value in a new row.
    - Ensure each column contains atomic (single, indivisible) values.
    - Identify a primary key for the table.
3. **Apply 2NF (Second Normal Form):**
    - Must be in 1NF.
    - Eliminate partial dependencies: Any non-key attribute must be fully dependent on the *entire* primary key. If a non-key attribute depends only on part of a composite primary key, move it to a new table with that part of the primary key as its own primary key.
4. **Apply 3NF (Third Normal Form):**
    - Must be in 2NF.
    - Eliminate transitive dependencies: Any non-key attribute must not be dependent on another non-key attribute. If it is, move the dependent attributes to a new table.
5. **Apply BCNF (Boyce-Codd Normal Form - Optional, but good for advanced study):**
    - Must be in 3NF.
    - Every determinant must be a candidate key. (A determinant is any attribute or set of attributes that determines another attribute). This addresses certain anomalies not caught by 3NF where a non-key attribute determines part of a composite key.
6. **Document Transformations:** For each step, show the original table, the transformed tables, and explain the reasons for the changes.

Source Code:

- Normalization is a conceptual design process, not directly "source code" in the sense of executable SQL.
- The "source code" here would be the SQL DDL statements that create the normalized tables.

```
-- Example: Unnormalized Table (Conceptual)
-- StudentCourse (StudentID, StudentName, StudentMajor, CourseID, CourseName,
InstructorName, InstructorEmail, Grade)
-- Problem: Redundancy (StudentName, StudentMajor, CourseName, InstructorName,
InstructorEmail repeated)
--           Partial Dependency (CourseName depends only on CourseID, not
StudentID + CourseID)
--           Transitive Dependency (InstructorEmail depends on InstructorName,
which depends on CourseID)

-- Step 1: Normalize to 1NF
-- Each cell contains a single value. A composite primary key (StudentID,
CourseID) is identified.
-- No SQL needed, just conceptual understanding.
```

```
-- Step 2: Normalize to 2NF (Eliminate Partial Dependencies)
-- Break down StudentCourse into:
-- Students (StudentID, StudentName, StudentMajor)
-- Courses (CourseID, CourseName, InstructorName, InstructorEmail) -- Still has
transitive dependency
-- Enrollments (StudentID, CourseID, Grade)

-- SQL for 2NF tables:
CREATE TABLE Students_2NF (
    StudentID INT PRIMARY KEY,
    StudentName VARCHAR(100),
    StudentMajor VARCHAR(100)
);

CREATE TABLE Courses_2NF (
    CourseID VARCHAR(10) PRIMARY KEY,
    CourseName VARCHAR(100),
    InstructorName VARCHAR(100), -- Transitive dependency on InstructorEmail
    InstructorEmail VARCHAR(100)
);

CREATE TABLE Enrollments_2NF (
    StudentID INT,
    CourseID VARCHAR(10),
    Grade CHAR(2),
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Students_2NF(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses_2NF(CourseID)
);

-- Step 3: Normalize to 3NF (Eliminate Transitive Dependencies)
-- Break down Courses_2NF into:
-- Courses (CourseID, CourseName, InstructorID)
-- Instructors (InstructorID, InstructorName, InstructorEmail)

-- SQL for 3NF tables:
CREATE TABLE Students_3NF (
    StudentID INT PRIMARY KEY,
    StudentName VARCHAR(100),
    StudentMajor VARCHAR(100)
);

CREATE TABLE Instructors_3NF (
    InstructorID INT PRIMARY KEY AUTO_INCREMENT,
    InstructorName VARCHAR(100) NOT NULL,
    InstructorEmail VARCHAR(100) UNIQUE
);

CREATE TABLE Courses_3NF (
    CourseID VARCHAR(10) PRIMARY KEY,
    CourseName VARCHAR(100) NOT NULL,
    InstructorID INT,
    FOREIGN KEY (InstructorID) REFERENCES Instructors_3NF(InstructorID)
);

CREATE TABLE Enrollments_3NF (
    StudentID INT,
    CourseID VARCHAR(10),
    Grade CHAR(2),
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Students_3NF(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses_3NF(CourseID)
);

-- DML: Sample data to illustrate normalization
```

```
-- (You would insert data into the normalized tables based on the original
unnormalized data)
INSERT INTO Students_3NF (StudentID, StudentName, StudentMajor) VALUES
(1, 'Alice', 'CS'), (2, 'Bob', 'EE');

INSERT INTO Instructors_3NF (InstructorName, InstructorEmail) VALUES
('Dr. Smith', 'smith@univ.edu'), ('Prof. Jones', 'jones@univ.edu');

INSERT INTO Courses_3NF (CourseID, CourseName, InstructorID) VALUES
('CS101', 'Intro to CS', 1), ('EE201', 'Circuits', 2);

INSERT INTO Enrollments_3NF (StudentID, CourseID, Grade) VALUES
(1, 'CS101', 'A'), (2, 'EE201', 'B');
```

Input: An example of an unnormalized table (conceptual) with data that exhibits redundancy and anomalies.

Expected Output: A set of normalized tables (e.g., in 3NF or BCNF) with their respective SQL `CREATE TABLE` statements, demonstrating how the original unnormalized data is broken down into smaller, related tables to improve data integrity and reduce redundancy. The explanation of each normalization step is crucial.

Lab 13: Case study submission for database administration

Title: Database Administration Case Study Submission

Aim: To analyze a given database administration (DBA) scenario, identify potential issues or requirements, and propose comprehensive solutions related to database management, performance, security, or maintenance.

Procedure:

1. **Understand the Case Study:** Thoroughly read the DBA case study, identifying the problem statement, existing infrastructure, performance bottlenecks, security concerns, or maintenance challenges.
2. **Identify Key Areas:** Categorize the issues into specific DBA domains (e.g., performance tuning, backup & recovery, security management, user access, storage management, patching).
3. **Analyze and Propose Solutions:** For each identified area:
    - **Problem Description:** Clearly articulate the specific problem.
    - **Proposed Solution:** Detail the steps, tools, and strategies to address the problem.
    - **Justification:** Explain why the proposed solution is appropriate, considering factors like cost, complexity, impact, and best practices.
    - **Implementation Steps:** Outline a high-level plan for implementing the solution.
    - **Monitoring/Verification:** Describe how the effectiveness of the solution would be monitored.
4. **Consider Best Practices:** Incorporate industry best practices for database administration.
5. **Structure the Report:** Organize your findings and proposals into a clear, well-structured report.

Source Code:

- This lab involves conceptual and strategic planning rather than direct executable code.
- "Source code" might refer to pseudo-code for administrative scripts (e.g., backup scripts, monitoring scripts) or configuration snippets (e.g., for database parameters, user roles).

```
-- Example Pseudo-code/SQL for a DBA Case Study Solution (e.g., for "Performance
Tuning")

-- Problem: Slow query on large table 'Orders'
-- Proposed Solution: Add an index to the 'OrderDate' column, used frequently in
WHERE clauses.

-- SQL (Conceptual):
-- Analyze current query performance (before index)
-- EXPLAIN SELECT * FROM Orders WHERE OrderDate BETWEEN '2023-01-01' AND '2023-
01-31';

-- Create index
CREATE INDEX idx_orders_orderdate ON Orders (OrderDate);

-- Re-analyze query performance (after index)
-- EXPLAIN SELECT * FROM Orders WHERE OrderDate BETWEEN '2023-01-01' AND '2023-
01-31';

-- Example Pseudo-code for a Backup Strategy:
-- Daily Full Backup to S3 at 2 AM
-- Weekly Differential Backup to S3 at 3 AM on Sundays
```

```
-- Transaction Log Backups every 15 minutes to local disk

-- Script (Conceptual):
-- #!/bin/bash
-- DB_NAME="production_db"
-- BACKUP_DIR="/mnt/backups/db"
-- TIMESTAMP=$(date +"%Y%m%d_%H%M%S")

-- # Full Backup
-- mysqldump -u user -p password $DB_NAME >
$BACKUP_DIR/$DB_NAME_full_$TIMESTAMP.sql
-- aws s3 cp $BACKUP_DIR/$DB_NAME_full_$TIMESTAMP.sql s3://my-db-
backups/$DB_NAME/full/

-- # Differential Backup (logic would be more complex, often using native DB
tools)
-- # SQL Server: BACKUP DATABASE MyDatabase TO DISK='...' WITH DIFFERENTIAL
-- # PostgreSQL: pg_basebackup -D /path/to/backup -Ft -Xf -P

-- # Transaction Log Backup (conceptual)
-- # SQL Server: BACKUP LOG MyDatabase TO DISK='...'
```

Input: A detailed description of a database administration scenario or problem.

Expected Output: A comprehensive report or presentation outlining:

- An analysis of the DBA case study.
- Identification of specific database administration challenges.
- Proposed solutions for each challenge, including:
    - Technical details (e.g., SQL commands, configuration changes, script logic).
    - Justification for the chosen solutions.
    - Considerations for implementation, monitoring, and potential risks.

Lab 14: Case study submission SLO-2 for recovery

Title: Database Recovery Case Study Submission (SLO-2)

Aim: To analyze a database failure scenario (e.g., hardware failure, data corruption, accidental deletion) and propose a detailed database recovery plan, ensuring data consistency and minimal downtime (meeting Service Level Objectives - SLOs).

Procedure:

1. **Understand the Failure Scenario:** Analyze the specific type of database failure described in the case study. Identify the extent of data loss or corruption, the time of failure, and any operational constraints.
2. **Identify Recovery Objectives:** Determine the RPO (Recovery Point Objective - how much data loss is acceptable) and RTO (Recovery Time Objective - how quickly the system must be restored) based on the case study's requirements.
3. **Assess Available Backups:** Identify what types of backups are available (full, differential, incremental, transaction logs) and their timestamps.
4. **Formulate Recovery Strategy:** Develop a step-by-step recovery plan:
    - **Pre-recovery steps:** Stop services, isolate the failed system.
    - **Restore latest full backup:** Restore the most recent full backup.
    - **Apply differential/incremental backups:** Apply any subsequent differential or incremental backups.
    - **Apply transaction logs:** Apply transaction log backups up to the point of failure (or desired recovery point).
    - **Post-recovery steps:** Verify data consistency, restart services, perform sanity checks.
5. **Consider Data Consistency:** Explain how the recovery process ensures data integrity and consistency.
6. **Document the Plan:** Present the recovery plan clearly, including commands, tools, and expected outcomes at each step. Discuss potential challenges and alternative approaches.

Source Code:

- This lab is primarily about strategic planning for disaster recovery.
- "Source code" would be the conceptual SQL/command-line commands for database restoration.

```
-- Example SQL/Command-line Pseudo-code for a Database Recovery Plan

-- Scenario: Production database server crashed due to disk failure.
-- RPO: 1 hour (max data loss). RTO: 4 hours (max downtime).
-- Backups:
--   - Full backup daily at 1 AM.
--   - Differential backup every 6 hours (7 AM, 1 PM, 7 PM).
--   - Transaction log backups every 15 minutes.
-- Failure occurred at 10:30 AM.

-- Recovery Plan Steps (Conceptual for a SQL Server-like environment):

-- 1. Prepare New Server/Disk:
--    - Provision new hardware or disk space.
--    - Install database software (e.g., SQL Server).

-- 2. Restore Latest Full Backup:
```

```
--    - Identify the latest full backup before failure (e.g., from 1 AM).
--    RESTORE DATABASE MyDatabase
--    FROM DISK = '/path/to/backup/MyDatabase_Full_20250521_010000.bak'
--    WITH NORECOVERY; -- Keep database in restoring state

-- 3. Restore Latest Differential Backup:
--    - Identify the latest differential backup before failure (e.g., from 7
AM).
--    RESTORE DATABASE MyDatabase
--    FROM DISK = '/path/to/backup/MyDatabase_Diff_20250521_070000.bak'
--    WITH NORECOVERY;

-- 4. Apply Transaction Log Backups:
--    - Apply all transaction log backups from the time of the differential
backup (7 AM) up to the point of failure (10:30 AM).
--    RESTORE LOG MyDatabase
--    FROM DISK = '/path/to/backup/MyDatabase_Log_20250521_071500.trn'
--    WITH NORECOVERY;
--    RESTORE LOG MyDatabase
--    FROM DISK = '/path/to/backup/MyDatabase_Log_20250521_073000.trn'
--    WITH NORECOVERY;
--    ... (continue for all logs up to 10:15 AM log)
--    RESTORE LOG MyDatabase
--    FROM DISK = '/path/to/backup/MyDatabase_Log_20250521_101500.trn'
--    WITH RECOVERY; -- Last log applies and brings database online

-- 5. Data Verification:
--    - Run integrity checks (e.g., DBCC CHECKDB).
--    - Perform sanity checks with application data.

-- 6. Restart Application Services:
--    - Bring the application online.
```

Input: A detailed description of a database failure scenario, including information about the type of failure, desired RPO/RTO, and available backup types/schedules.

Expected Output: A comprehensive database recovery plan document that includes:

- Analysis of the failure and its impact.
- Defined RPO and RTO.
- A step-by-step recovery procedure with specific commands or pseudo-code.
- Explanation of how data consistency is maintained.
- Discussion of verification steps and potential challenges.

Lab 15: Case study submission for database backups

Title: Database Backups Case Study Submission

Aim: To analyze a given business scenario and design a robust database backup strategy that meets specific business requirements for data protection, recovery time, and recovery point, considering various backup types and storage solutions.

Procedure:

1. **Understand Business Requirements:** Analyze the case study to determine:
    - **Data Criticality:** Which data is most important?
    - **RPO (Recovery Point Objective):** Maximum acceptable data loss (e.g., 1 hour, 24 hours).
    - **RTO (Recovery Time Objective):** Maximum acceptable downtime (e.g., 4 hours, 8 hours).
    - **Compliance/Retention:** Any legal or regulatory requirements for data retention.
    - **Budget/Resources:** Available budget, storage, and network bandwidth.
2. **Choose Backup Types:** Select appropriate backup types (Full, Differential, Incremental, Transaction Log) based on RPO/RTO and resource constraints.
3. **Define Backup Schedule:** Determine the frequency and timing for each backup type (e.g., daily full, hourly transaction logs).
4. **Select Storage Location:** Decide on backup storage locations (e.g., local disk, network share, cloud storage like S3, Azure Blob Storage). Consider redundancy and offsite storage.
5. **Implement Retention Policy:** Define how long backups should be kept.
6. **Develop Monitoring and Alerting:** Plan how backup success/failure will be monitored and how alerts will be generated.
7. **Plan for Testing:** Emphasize the importance of regularly testing backups to ensure recoverability.
8. **Document the Strategy:** Present the complete backup strategy in a clear and detailed report.

Source Code:

- This lab focuses on strategic planning for backup, not executable code.
- "Source code" would be conceptual commands or scripts for performing backups.

```
-- Example SQL/Command-line Pseudo-code for a Database Backup Strategy

-- Scenario: E-commerce database. High transaction volume.
-- RPO: 15 minutes. RTO: 2 hours.
-- Data Retention: 7 days of daily backups, 1 month of weekly full backups.

-- Proposed Backup Strategy:

-- 1. Full Backups:
--    - Frequency: Once daily, during off-peak hours (e.g., 2:00 AM).
--    - Command (Conceptual - MySQL):
--      mysqldump -u user -p password my_ecommerce_db >
/mnt/backups/daily/my_ecommerce_db_full_$(date +"%Y%m%d").sql
--    - Storage: Local NAS for 7 days, then offloaded to S3 for 1 month
retention.

-- 2. Transaction Log Backups (for point-in-time recovery):
--    - Frequency: Every 15 minutes.
```

```
--    - Command (Conceptual - SQL Server):
--      BACKUP LOG my_ecommerce_db TO DISK =
'/mnt/backups/logs/my_ecommerce_db_log_$(date +"%Y%m%d%H%M").trn'
--    - Storage: Local disk for 24 hours, then archived to S3 for 7 days.

-- 3. Differential Backups (Optional, for faster recovery between fulls and
logs):
--    - Frequency: Every 6 hours (e.g., 8 AM, 2 PM, 8 PM).
--    - Command (Conceptual - SQL Server):
--      BACKUP DATABASE my_ecommerce_db TO DISK =
'/mnt/backups/diff/my_ecommerce_db_diff_$(date +"%Y%m%d%H%M").bak' WITH
DIFFERENTIAL
--    - Storage: Local NAS for 24 hours.

-- 4. Backup Retention Policy:
--    - Daily Full Backups: Keep for 7 days on NAS, 30 days on S3.
--    - Transaction Log Backups: Keep for 24 hours locally, 7 days on S3.
--    - Differential Backups: Keep for 24 hours locally.

-- 5. Monitoring and Alerting:
--    - Implement cron jobs/scheduled tasks to run backups.
--    - Configure monitoring tools (e.g., Nagios, Zabbix) to check backup job
status.
--    - Set up email/SMS alerts for backup failures.

-- 6. Backup Testing:
--    - Schedule monthly recovery tests on a separate staging environment using
random backup sets.
--    - Document recovery times and verify data integrity.
```

Input: A detailed description of a business scenario requiring a database backup strategy, including details on data criticality, RPO, RTO, and any specific constraints.

Expected Output: A comprehensive database backup strategy document that includes:

- Analysis of business requirements (RPO, RTO, retention).
- Chosen backup types and their rationale.
- Detailed backup schedule.
- Storage locations and considerations.
- Retention policy.
- Plans for monitoring, alerting, and regular backup testing.