SRM Institute of Science and Technology

Department of Computer Applications

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

Circular – 2020-21

MCA GAI 1st semester

Advanced Database Technology (PGI20C02J)

Lab Manual

Lab 1: Create a Database Schema for University Database

Title: Creating a Database Schema for a University Database

Aim: To design and implement a relational database schema for a university system, including tables for students, courses, departments, and faculty, along with appropriate data types and relationships.

Procedure:

1. **Understand Requirements:** Analyze the typical entities and relationships within a university environment (e.g., Students enroll in Courses, Faculty teach Courses, Departments offer Courses).
2. **Entity-Relationship (ER) Modeling (Conceptual Design):** Identify entities (e.g., Student, Course, Department, Faculty), attributes for each entity, and relationships between them. Determine primary and foreign keys.
3. **Relational Schema Mapping (Logical Design):** Translate the ER model into a relational schema, defining tables, columns, data types, and constraints (primary key, foreign key, not null).
4. **SQL DDL Implementation (Physical Design):** Write SQL Data Definition Language (DDL) commands (CREATE TABLE) to create the defined tables in a chosen database system (e.g., MySQL, PostgreSQL, Oracle).
5. **Verification:** Confirm that tables are created correctly and relationships are established.

Source Code:

```
-- Create Department Table
CREATE TABLE Department (
    DeptID VARCHAR(10) PRIMARY KEY,
    DeptName VARCHAR(50) NOT NULL UNIQUE,
    Location VARCHAR(50)
);

-- Create Faculty Table
CREATE TABLE Faculty (
    FacultyID VARCHAR(10) PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE,
    HireDate DATE,
    DeptID VARCHAR(10),
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
);
```

```
-- Create Course Table
CREATE TABLE Course (
    CourseID VARCHAR(10) PRIMARY KEY,
    CourseName VARCHAR(100) NOT NULL,
    Credits INT NOT NULL CHECK (Credits > 0),
    DeptID VARCHAR(10),
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
);

-- Create Student Table
CREATE TABLE Student (
    StudentID VARCHAR(10) PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DateOfBirth DATE,
    Email VARCHAR(100) UNIQUE,
    MajorDeptID VARCHAR(10),
    FOREIGN KEY (MajorDeptID) REFERENCES Department(DeptID)
);

-- Create Enrollment Table (Many-to-Many relationship between Student and
Course)
CREATE TABLE Enrollment (
    EnrollmentID INT PRIMARY KEY AUTO_INCREMENT, -- For MySQL, use SERIAL for
PostgreSQL
    StudentID VARCHAR(10),
    CourseID VARCHAR(10),
    EnrollmentDate DATE DEFAULT CURRENT_DATE,
    Grade VARCHAR(2),
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID),
    UNIQUE (StudentID, CourseID) -- Ensures a student enrolls in a course only
once
);

-- Create Teaches Table (Many-to-Many relationship between Faculty and Course)
CREATE TABLE Teaches (
    TeachesID INT PRIMARY KEY AUTO_INCREMENT, -- For MySQL, use SERIAL for
PostgreSQL
    FacultyID VARCHAR(10),
    CourseID VARCHAR(10),
    Semester VARCHAR(20),
    Year INT,
    FOREIGN KEY (FacultyID) REFERENCES Faculty(FacultyID),
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID),
    UNIQUE (FacultyID, CourseID, Semester, Year) -- Ensures a faculty teaches a
specific course in a specific semester/year only once
);
```

Input: No direct input is required for schema creation, as it's a set of DDL commands. The "input" is the SQL script itself.

Expected Output: Successful execution of the SQL script, indicated by messages from the database system like "Query OK", "Tables created successfully", or similar. The database will contain the defined tables with their respective columns and constraints.

Lab 2: Create ER Model University Database

Title: Creating an Entity-Relationship (ER) Model for a University Database

Aim: To conceptually design a university database system using the Entity-Relationship (ER) modeling approach, identifying entities, attributes, and relationships.

Procedure:

1. **Identify Entities:** Determine the main objects or concepts in the university system that need to store data (e.g., Student, Course, Department, Faculty, Enrollment).
2. **Identify Attributes:** For each identified entity, list the relevant properties or characteristics (e.g., for Student: StudentID, Name, DateOfBirth, Email).
3. **Define Primary Keys:** Choose a unique identifier for each entity.
4. **Identify Relationships:** Determine how entities are associated with each other (e.g., "Student enrolls in Course", "Faculty teaches Course", "Department offers Course").
5. **Determine Cardinality and Participation:** Specify the number of instances of one entity that can be associated with instances of another entity (e.g., one-to-one, one-to-many, many-to-many) and whether participation is optional or mandatory.
6. **Draw ER Diagram:** Represent the entities, attributes, and relationships graphically using standard ER diagram notation (rectangles for entities, ovals for attributes, diamonds for relationships, lines for connections). This can be done using tools like Lucidchart, draw.io, or even by hand.

Source Code: *This lab focuses on conceptual design, so there is no direct "source code" in the programming sense. The output is typically an ER Diagram.*

Input: Conceptual requirements for a university database system.

Expected Output: A well-formed Entity-Relationship (ER) Diagram representing the university database, clearly showing:

- Entities (e.g., Student, Course, Department, Faculty)
- Attributes for each entity, with primary keys underlined.
- Relationships between entities (e.g., ENROLLS, TEACHES, OFFERS).
- Cardinality (e.g., 1:N, M:N) and participation constraints for each relationship.

Lab 3: Implement Integrity Constraints

Title: Implementing Integrity Constraints in a Relational Database

Aim: To understand and apply various types of integrity constraints (Primary Key, Foreign Key, NOT NULL, UNIQUE, CHECK) to maintain data accuracy and consistency in a database.

Procedure:

1. **Review Schema:** Use the university database schema created in Lab 1.
2. **Primary Key Constraint:** Ensure each table has a primary key defined to uniquely identify records.
3. **Foreign Key Constraint:** Establish referential integrity between related tables using foreign keys to ensure valid relationships.
4. **NOT NULL Constraint:** Apply to columns that must always contain a value.
5. **UNIQUE Constraint:** Apply to columns where all values must be distinct (e.g., email addresses).
6. **CHECK Constraint:** Define rules that data in a column must satisfy (e.g., credits for a course must be positive).
7. **Test Constraints:** Attempt to insert, update, or delete data that violates each type of constraint to observe the database's error handling.

Source Code:

```
-- Assuming tables from Lab 1 are already created.
-- Example of adding constraints (if not already defined during CREATE TABLE)

-- Add NOT NULL constraint to CourseName if not present
ALTER TABLE Course
ALTER COLUMN CourseName SET NOT NULL;

-- Add UNIQUE constraint to Faculty Email if not present
ALTER TABLE Faculty
ADD CONSTRAINT UQ_FacultyEmail UNIQUE (Email);

-- Add CHECK constraint to Credits in Course table if not present
ALTER TABLE Course
ADD CONSTRAINT CHK_CourseCredits CHECK (Credits > 0);

-- Test cases for constraint violations:

-- 1. Primary Key Violation (e.g., inserting duplicate StudentID)
INSERT INTO Student (StudentID, FirstName, LastName, DateOfBirth, Email,
MajorDeptID)
VALUES ('S001', 'John', 'Doe', '2000-01-15', 'john.doe@example.com', 'CS');
-- Attempting to insert 'S001' again will fail if S001 already exists.

-- 2. Foreign Key Violation (e.g., enrolling a student in a non-existent course)
INSERT INTO Enrollment (StudentID, CourseID)
VALUES ('S001', 'NONEXISTENT_COURSE'); -- This should fail

-- 3. NOT NULL Violation (e.g., inserting a course without a name)
INSERT INTO Course (CourseID, CourseName, Credits, DeptID)
VALUES ('C005', NULL, 3, 'CS'); -- This should fail

-- 4. UNIQUE Violation (e.g., inserting faculty with duplicate email)
INSERT INTO Faculty (FacultyID, FirstName, LastName, Email, HireDate, DeptID)
VALUES ('F003', 'Jane', 'Smith', 'jane.doe@example.com', '2020-09-01', 'EE');
```

```
-- If jane.doe@example.com already exists for F001, this will fail.

-- 5. CHECK Constraint Violation (e.g., inserting a course with negative
credits)
INSERT INTO Course (CourseID, CourseName, Credits, DeptID)
VALUES ('C006', 'Advanced Topics', -1, 'CS'); -- This should fail
```

Input: SQL DDL commands for adding constraints and SQL DML commands for testing constraint violations.

Expected Output:

- Successful execution messages for DDL commands.
- Error messages from the database system when attempting to violate constraints, indicating that the constraints are correctly enforced. For example, "Duplicate entry for key 'PRIMARY'", "Cannot add or update a child row: a foreign key constraint fails", "Column 'CourseName' cannot be null", etc.

Lab 4: Implement DDL, DML commands

Title: Implementing Data Definition Language (DDL) and Data Manipulation Language (DML) Commands

Aim: To gain practical experience with DDL commands for defining and managing database objects and DML commands for manipulating data within those objects.

Procedure:

1. **DDL Commands (CREATE, ALTER, DROP, TRUNCATE, RENAME):**
    - `CREATE TABLE`: Create a new table (already done in Lab 1).
    - `ALTER TABLE`: Add a new column, modify a column's data type, add/drop constraints.
    - `DROP TABLE`: Delete an existing table.
    - `TRUNCATE TABLE`: Remove all rows from a table, but keep the table structure.
    - `RENAME TABLE`: Change the name of a table.
2. **DML Commands (INSERT, SELECT, UPDATE, DELETE):**
    - `INSERT INTO`: Add new rows of data into tables.
    - `SELECT`: Retrieve data from tables, including filtering, ordering, and aggregation.
    - `UPDATE`: Modify existing data in tables.
    - `DELETE FROM`: Remove rows from tables.
3. **Practice Scenarios:** Execute various combinations of DDL and DML commands to understand their effects on the database schema and data.

Source Code:

```
-- DDL Examples (assuming Department table exists)

-- Add a new column 'Budget' to Department table
ALTER TABLE Department
ADD COLUMN Budget DECIMAL(10, 2);

-- Modify the data type of Location column in Department
-- (Note: This might require dropping and re-adding for some DBs if data exists)
ALTER TABLE Department
MODIFY COLUMN Location VARCHAR(100); -- For MySQL
-- ALTER TABLE Department ALTER COLUMN Location TYPE VARCHAR(100); -- For
PostgreSQL

-- Drop the Budget column
ALTER TABLE Department
DROP COLUMN Budget;

-- Rename the Department table
RENAME TABLE Department TO UniversityDepartment; -- For MySQL
-- ALTER TABLE Department RENAME TO UniversityDepartment; -- For PostgreSQL

-- Revert table name for consistency with other labs
RENAME TABLE UniversityDepartment TO Department;

-- DML Examples (assuming data is present or will be inserted)

-- Insert data into Department table
INSERT INTO Department (DeptID, DeptName, Location)
VALUES ('CS', 'Computer Science', 'Building A'),
       ('EE', 'Electrical Engineering', 'Building B'),
```

```
       ('ME', 'Mechanical Engineering', 'Building C');

-- Insert data into Student table
INSERT INTO Student (StudentID, FirstName, LastName, DateOfBirth, Email,
MajorDeptID)
VALUES ('S001', 'Alice', 'Smith', '2001-05-20', 'alice.s@example.com', 'CS'),
       ('S002', 'Bob', 'Johnson', '2002-03-10', 'bob.j@example.com', 'EE');

-- Select all data from Student table
SELECT * FROM Student;

-- Select specific columns from Course table
SELECT CourseID, CourseName, Credits FROM Course;

-- Update a student's email
UPDATE Student
SET Email = 'alice.smith@example.com'
WHERE StudentID = 'S001';

-- Delete a department (ensure no foreign key dependencies, or use CASCADE)
-- DELETE FROM Department WHERE DeptID = 'ME'; -- This might fail if
students/courses are linked

-- Truncate the Enrollment table (removes all data, keeps structure)
-- TRUNCATE TABLE Enrollment;
```

Input: SQL DDL and DML commands.

Expected Output:

- **DDL:** Messages indicating successful schema modifications (e.g., "Query OK", "Table altered").
- **DML:**
  - `INSERT`: Messages indicating rows inserted (e.g., "1 row affected").
  - `SELECT`: Display of queried data in a tabular format.
  - `UPDATE`: Messages indicating rows updated (e.g., "1 row affected").
  - `DELETE`: Messages indicating rows deleted (e.g., "X rows affected").
  - `TRUNCATE`: Message indicating table truncated.

Lab 5: Implement DCL, TCL

Title: Implementing Data Control Language (DCL) and Transaction Control Language (TCL)

Aim: To understand and apply DCL commands for managing database security (permissions) and TCL commands for controlling transactions (ensuring atomicity, consistency, isolation, durability).

Procedure:

1. **DCL Commands (GRANT, REVOKE):**
    - Create a new database user.
    - `GRANT`: Assign specific privileges (e.g., SELECT, INSERT, UPDATE, DELETE) on tables to users.
    - `REVOKE`: Remove previously granted privileges from users.
    - Test by logging in as the new user and attempting privileged operations.
2. **TCL Commands (COMMIT, ROLLBACK, SAVEPOINT):**
    - Start a transaction (implicitly or explicitly depending on DB system).
    - Perform a series of DML operations (e.g., inserts, updates).
    - `COMMIT`: Make all changes in the transaction permanent.
    - `ROLLBACK`: Undo all changes made since the beginning of the transaction or the last `SAVEPOINT`.
    - `SAVEPOINT`: Set a point within a transaction to which you can later roll back.
    - Demonstrate scenarios where `COMMIT` and `ROLLBACK` are used to maintain data integrity.

Source Code:

```
-- DCL Examples (Syntax might vary slightly between DB systems)

-- Create a new user (example for MySQL)
CREATE USER 'lab_user'@'localhost' IDENTIFIED BY 'password123';

-- Grant SELECT and INSERT privileges on the Student table to 'lab_user'
GRANT SELECT, INSERT ON University.Student TO 'lab_user'@'localhost';

-- Grant all privileges on the Course table to 'lab_user'
GRANT ALL PRIVILEGES ON University.Course TO 'lab_user'@'localhost';

-- Revoke INSERT privilege on Student table from 'lab_user'
REVOKE INSERT ON University.Student FROM 'lab_user'@'localhost';

-- Drop the user (cleanup)
DROP USER 'lab_user'@'localhost';


-- TCL Examples (assuming auto-commit is off or explicit transaction block)

-- Scenario 1: Successful Transaction
START TRANSACTION; -- For MySQL/PostgreSQL
-- BEGIN TRANSACTION; -- For SQL Server/Oracle

INSERT INTO Student (StudentID, FirstName, LastName, DateOfBirth, Email,
MajorDeptID)
VALUES ('S003', 'Charlie', 'Brown', '2003-11-01', 'charlie.b@example.com',
'CS');

UPDATE Course
```

```
SET Credits = 4
WHERE CourseID = 'C001';

COMMIT; -- Makes changes permanent

-- Scenario 2: Transaction with Rollback
START TRANSACTION;

INSERT INTO Faculty (FacultyID, FirstName, LastName, Email, HireDate, DeptID)
VALUES ('F004', 'Diana', 'Prince', 'diana.p@example.com', '2022-01-01', 'EE');

-- Introduce an error or decide to undo
-- For example, trying to insert duplicate primary key or a value that violates
a constraint
-- INSERT INTO Faculty (FacultyID, FirstName, LastName, Email, HireDate, DeptID)
-- VALUES ('F004', 'Diana', 'Prince', 'diana.p@example.com', '2022-01-01',
'EE'); -- This would cause an error

ROLLBACK; -- Undoes all changes since START TRANSACTION

-- Scenario 3: Using SAVEPOINT
START TRANSACTION;

INSERT INTO Department (DeptID, DeptName, Location) VALUES ('PHY', 'Physics',
'Building D');

SAVEPOINT after_dept_insert;

INSERT INTO Course (CourseID, CourseName, Credits, DeptID) VALUES ('C007',
'Quantum Mechanics', 3, 'PHY');

-- Decide to rollback only the course insert
ROLLBACK TO SAVEPOINT after_dept_insert;

-- Now, only the Department 'PHY' exists, 'C007' is rolled back.
COMMIT; -- Commits the Department insert
```

Input: SQL DCL and TCL commands.

Expected Output:

- **DCL:** Successful execution messages for `GRANT` and `REVOKE`. When testing, observe permission denied errors when a user tries to perform unauthorized actions, and successful execution for authorized actions.
- **TCL:**
  - After `COMMIT`, data changes are persistent and visible to other sessions.
  - After `ROLLBACK`, data changes are undone and not visible.
  - After `ROLLBACK TO SAVEPOINT`, changes up to the savepoint are preserved, and subsequent changes are undone.

Lab 6: Implement SQL subqueries, Joins and Clauses

Title: Implementing SQL Subqueries, Joins, and Clauses for Data Retrieval

Aim: To master advanced SQL query techniques, including subqueries for nested queries, various types of joins for combining data from multiple tables, and clauses (WHERE, GROUP BY, HAVING, ORDER BY) for filtering, grouping, and sorting results.

Procedure:

1. **Subqueries:**
    - Write queries using `IN`, `EXISTS`, `NOT IN`, `NOT EXISTS` with subqueries.
    - Implement correlated and non-correlated subqueries.
    - Use subqueries in `SELECT`, `FROM`, and `WHERE` clauses.
2. **Joins:**
    - Perform `INNER JOIN` to retrieve matching rows from two or more tables.
    - Perform `LEFT JOIN` (or `LEFT OUTER JOIN`) to include all rows from the left table.
    - Perform `RIGHT JOIN` (or `RIGHT OUTER JOIN`) to include all rows from the right table.
    - Perform `FULL OUTER JOIN` (if supported by DB, or simulate with UNION) to include all rows from both tables.
    - Implement `SELF JOIN` for relationships within the same table.
3. **Clauses:**
    - `WHERE`: Filter rows based on specified conditions.
    - `GROUP BY`: Group rows that have the same values in specified columns into summary rows.
    - `HAVING`: Filter groups based on specified conditions.
    - `ORDER BY`: Sort the result set in ascending or descending order.
4. **Complex Queries:** Combine subqueries, joins, and clauses to answer complex business questions related to the university database.

Source Code:

```
-- Assuming data is populated in the university database tables

-- Subquery Examples

-- 1. Students enrolled in 'Computer Science' department courses (using IN)
SELECT S.FirstName, S.LastName
FROM Student S
WHERE S.MajorDeptID IN (SELECT DeptID FROM Department WHERE DeptName = 'Computer
Science');

-- 2. Courses with more than 3 credits (using subquery in FROM clause for
aggregation)
SELECT DeptName, AVG_Credits
FROM (
    SELECT DeptID, AVG(Credits) AS AVG_Credits
    FROM Course
    GROUP BY DeptID
) AS DeptAvgCredits
JOIN Department D ON DeptAvgCredits.DeptID = D.DeptID
WHERE AVG_Credits > 3;

-- 3. Faculty who teach at least one course (using EXISTS)
SELECT F.FirstName, F.LastName
```

```
FROM Faculty F
WHERE EXISTS (SELECT 1 FROM Teaches T WHERE T.FacultyID = F.FacultyID);


-- Join Examples

-- 1. INNER JOIN: Students and their enrolled courses
SELECT S.FirstName, S.LastName, C.CourseName
FROM Student S
INNER JOIN Enrollment E ON S.StudentID = E.StudentID
INNER JOIN Course C ON E.CourseID = C.CourseID;

-- 2. LEFT JOIN: All departments and the courses they offer (including
departments with no courses)
SELECT D.DeptName, C.CourseName
FROM Department D
LEFT JOIN Course C ON D.DeptID = C.DeptID;

-- 3. RIGHT JOIN: All courses and the departments offering them (including
courses not linked to a department)
-- (Often converted to LEFT JOIN by swapping tables for consistency)
SELECT D.DeptName, C.CourseName
FROM Course C
RIGHT JOIN Department D ON C.DeptID = D.DeptID;

-- 4. SELF JOIN: Find students who share the same major department
SELECT S1.FirstName AS Student1_FirstName, S1.LastName AS Student1_LastName,
       S2.FirstName AS Student2_FirstName, S2.LastName AS Student2_LastName,
       D.DeptName
FROM Student S1
INNER JOIN Student S2 ON S1.MajorDeptID = S2.MajorDeptID AND S1.StudentID <>
S2.StudentID
INNER JOIN Department D ON S1.MajorDeptID = D.DeptID
ORDER BY D.DeptName, S1.LastName, S2.LastName;


-- Clause Examples

-- 1. WHERE: Students born after 2000
SELECT StudentID, FirstName, LastName
FROM Student
WHERE DateOfBirth > '2000-12-31';

-- 2. GROUP BY and HAVING: Count students per department, only show departments
with more than 1 student
SELECT D.DeptName, COUNT(S.StudentID) AS NumberOfStudents
FROM Department D
JOIN Student S ON D.DeptID = S.MajorDeptID
GROUP BY D.DeptName
HAVING COUNT(S.StudentID) > 1;

-- 3. ORDER BY: All courses, ordered by credits (descending) and then by course
name (ascending)
SELECT CourseID, CourseName, Credits
FROM Course
ORDER BY Credits DESC, CourseName ASC;
```

Input: SQL `SELECT` statements incorporating subqueries, joins, and various clauses.

Expected Output: Tabular results displaying the data retrieved by each query, correctly filtered, joined, grouped, and ordered according to the specified conditions.

Lab 7: Implementing PL/SQL Conditional Statements, Looping Statements

Title: Implementing PL/SQL Conditional and Looping Statements

Aim: To learn how to write procedural SQL (PL/SQL in Oracle, or equivalent in other databases like T-SQL for SQL Server, pg/SQL for PostgreSQL) using conditional logic (IF-THEN-ELSE) and iterative constructs (LOOP, WHILE, FOR) to perform more complex database operations.

Procedure:

1. **Setup Environment:** Ensure a PL/SQL (or equivalent) environment is available (e.g., Oracle SQL Developer, SQL*Plus).
2. **Conditional Statements (IF-THEN-ELSE, ELSIF):**
    - Write PL/SQL blocks that execute different code paths based on conditions.
    - Use `IF-THEN`, `IF-THEN-ELSE`, and `IF-THEN-ELSIF-ELSE` structures.
3. **Looping Statements (LOOP, WHILE, FOR):**
    - Implement basic `LOOP` with `EXIT WHEN` for controlled iteration.
    - Use `WHILE` loops for pre-condition controlled iteration.
    - Utilize `FOR` loops, especially cursor-based `FOR` loops for processing query results row by row.
4. **Practical Scenarios:** Develop PL/SQL blocks to automate tasks like updating student grades based on a score, generating reports, or performing data validation.

Source Code:

```
-- PL/SQL Examples (Oracle syntax)

-- 1. Conditional Statement (IF-THEN-ELSE)
DECLARE
    v_student_id VARCHAR(10) := 'S001';
    v_grade      VARCHAR(2);
BEGIN
    -- Assume a function or query to get the grade for S001
    -- For demonstration, let's hardcode a grade
    v_grade := 'A';

    IF v_grade = 'A' THEN
        DBMS_OUTPUT.PUT_LINE('Student ' || v_student_id || ' achieved an
excellent grade.');
    ELSIF v_grade = 'B' THEN
        DBMS_OUTPUT.PUT_LINE('Student ' || v_student_id || ' achieved a good
grade.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Student ' || v_student_id || ' needs
improvement.');
    END IF;
END;
/

-- 2. Looping Statement (FOR Loop - Numeric)
DECLARE
    v_counter NUMBER := 0;
BEGIN
    FOR i IN 1..5 LOOP
        v_counter := v_counter + 1;
```

```
        DBMS_OUTPUT.PUT_LINE('Loop iteration: ' || i || ', Counter: ' ||
v_counter);
    END LOOP;
END;
/

-- 3. Looping Statement (WHILE Loop)
DECLARE
    v_num NUMBER := 1;
BEGIN
    WHILE v_num <= 3 LOOP
        DBMS_OUTPUT.PUT_LINE('Current number: ' || v_num);
        v_num := v_num + 1;
    END LOOP;
END;
/

-- 4. Cursor-based FOR Loop (processing students)
DECLARE
    CURSOR student_cursor IS
        SELECT StudentID, FirstName, LastName, MajorDeptID
        FROM Student
        WHERE MajorDeptID = 'CS';
BEGIN
    DBMS_OUTPUT.PUT_LINE('--- Computer Science Students ---');
    FOR s_rec IN student_cursor LOOP
        DBMS_OUTPUT.PUT_LINE('ID: ' || s_rec.StudentID || ', Name: ' ||
s_rec.FirstName || ' ' || s_rec.LastName);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('-------------------------------');
END;
/
```

Input: PL/SQL code blocks executed in a SQL client.

Expected Output: Output printed to the console or log, demonstrating the flow control:

- For conditional statements: The appropriate message based on the condition.
- For loops: Iterated messages or data processed row by row.

Lab 8: Write a program to implement PL/SQL functions

Title: Implementing PL/SQL Functions

Aim: To create and utilize PL/SQL functions to encapsulate reusable logic, perform calculations, and return a single value, thereby improving code modularity and maintainability.

Procedure:

1. **Understand Functions:** Differentiate between procedures (which perform actions) and functions (which compute and return a value).
2. **Function Definition:** Define a PL/SQL function with parameters (if any) and a `RETURN` type.
3. **Function Body:** Implement the logic within the function to perform the desired computation.
4. **Function Usage:** Call the function within SQL queries (e.g., in `SELECT` or `WHERE` clauses) or within other PL/SQL blocks.
5. **Error Handling:** Implement exception handling within functions using `EXCEPTION` blocks.

Source Code:

```
-- PL/SQL Function Examples (Oracle syntax)

-- 1. Function to calculate the age of a student
CREATE OR REPLACE FUNCTION CalculateStudentAge (p_dob IN DATE)
RETURN NUMBER
IS
    v_age NUMBER;
BEGIN
    v_age := TRUNC(MONTHS_BETWEEN(SYSDATE, p_dob) / 12);
    RETURN v_age;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error calculating age: ' || SQLERRM);
        RETURN NULL; -- Or raise an application-specific error
END;
/

-- Test the CalculateStudentAge function
SELECT StudentID, FirstName, LastName, DateOfBirth,
CalculateStudentAge(DateOfBirth) AS Age
FROM Student;

-- 2. Function to get the full name of a faculty member
CREATE OR REPLACE FUNCTION GetFacultyFullName (p_faculty_id IN VARCHAR2)
RETURN VARCHAR2
IS
    v_full_name VARCHAR2(100);
BEGIN
    SELECT FirstName || ' ' || LastName
    INTO v_full_name
    FROM Faculty
    WHERE FacultyID = p_faculty_id;

    RETURN v_full_name;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Faculty ID ' || p_faculty_id || ' not found.');
```

```
        RETURN NULL;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error getting faculty full name: ' || SQLERRM);
        RETURN NULL;
END;
/

-- Test the GetFacultyFullName function
SELECT CourseID, GetFacultyFullName(T.FacultyID) AS InstructorName
FROM Teaches T
WHERE Semester = 'Fall' AND Year = 2024; -- Assuming some data in Teaches table
```

Input: PL/SQL function definitions and SQL queries/PL/SQL blocks that call these functions.

Expected Output:

- Successful compilation messages for the functions.
- When functions are called in `SELECT` statements, the computed values (e.g., age, full name) will appear as new columns in the result set.
- Error messages if exceptions are raised within the function.

Lab 9: Study the structure and properties of B-tree index and its variants

Title: Studying the Structure and Properties of B-tree Indexes and Their Variants

Aim: To understand the internal structure, operational principles, and performance implications of B-tree indexes, including how they facilitate efficient data retrieval and their variants.

Procedure:

1. **Theoretical Study:**
    - Research the concept of indexing in databases.
    - Learn about the B-tree data structure: nodes, keys, pointers, fanout, balanced property.
    - Understand how B-trees are used for efficient searching, insertion, and deletion.
    - Explore the concept of "height" of a B-tree and its impact on performance.
2. **Practical Demonstration (Conceptual/Simulated):**
    - **Create an Index:** Use `CREATE INDEX` command on a large table (e.g., Student or Enrollment) on a non-primary key column.
    - **Analyze Query Performance:**
        - Run `SELECT` queries on the indexed column with and without the index (by dropping and recreating, or using hints if available).
        - Use `EXPLAIN PLAN` (or similar tool like `EXPLAIN ANALYZE` in PostgreSQL, `EXPLAIN` in MySQL) to view the query execution plan and observe how the optimizer uses the index.
    - **Observe Index Maintenance:** Understand how inserts, updates, and deletes affect the B-tree structure (splitting, merging nodes). (This is typically observed through performance changes rather than direct visual inspection of the tree).
3. **Variants Discussion:** Discuss other index types like B+ trees (most common in relational databases), hash indexes, bitmap indexes, and their specific use cases.

Source Code:

```
-- Example for MySQL (similar commands exist for other DBs)

-- Create a large table for demonstration (if Student table is small)
CREATE TABLE LargeStudent (
    StudentID VARCHAR(10) PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Email VARCHAR(100) UNIQUE,
    MajorDeptID VARCHAR(10)
);

-- Insert a large number of rows (e.g., 100,000 or more)
-- This would typically be done via a script or bulk insert
-- Example:
-- INSERT INTO LargeStudent (StudentID, FirstName, LastName, Email, MajorDeptID)
-- SELECT
--     LPAD(ROW_NUMBER() OVER (ORDER BY (SELECT NULL)), 10, 'S'),
--     CONCAT('FirstName', FLOOR(RAND() * 1000)),
--     CONCAT('LastName', FLOOR(RAND() * 1000)),
--     CONCAT('email', ROW_NUMBER() OVER (ORDER BY (SELECT NULL)),
'@example.com'),
--     CASE FLOOR(RAND() * 3) WHEN 0 THEN 'CS' WHEN 1 THEN 'EE' ELSE 'ME' END
```

```
-- FROM information_schema.tables LIMIT 100000; -- Adjust LIMIT as needed

-- Query without index (assuming no index on LastName)
SELECT * FROM LargeStudent WHERE LastName = 'LastName500';

-- Create a B-tree index on LastName
CREATE INDEX idx_lastname ON LargeStudent (LastName);

-- Query with index (should be faster)
SELECT * FROM LargeStudent WHERE LastName = 'LastName500';

-- Analyze query plan (MySQL)
EXPLAIN SELECT * FROM LargeStudent WHERE LastName = 'LastName500';

-- Drop the index (for comparison or cleanup)
DROP INDEX idx_lastname ON LargeStudent;

-- Re-analyze query plan without index
EXPLAIN SELECT * FROM LargeStudent WHERE LastName = 'LastName500';
```

Input:

- Theoretical concepts and diagrams of B-trees.
- SQL `CREATE INDEX`, `DROP INDEX`, and `SELECT` statements.
- `EXPLAIN PLAN` or `EXPLAIN` commands.

Expected Output:

- **Theoretical:** Diagrams illustrating B-tree structure, explanations of its properties (balanced, fanout, search path).
- **Practical:**
  - Successful index creation messages.
  - Observation of faster query execution times when an appropriate index is used.
  - Query execution plans showing "Using index" or "Index scan" when the index is utilized, and "Full table scan" when it's not.

Lab 10: Write functions/procedures to begin, commit, and rollback transactions.

Title: Implementing Functions/Procedures for Transaction Control

Aim: To develop PL/SQL (or equivalent) functions or procedures that explicitly manage database transactions using `BEGIN`, `COMMIT`, and `ROLLBACK` commands, ensuring data integrity and atomicity.

Procedure:

1. **Review TCL:** Revisit the concepts of `COMMIT` and `ROLLBACK` from Lab 5.
2. **Create Procedure for Data Insertion with Commit:**
    - Write a procedure that inserts multiple records into a table.
    - Include `COMMIT` at the end to make the changes permanent.
3. **Create Procedure for Data Insertion with Rollback on Error:**
    - Write a procedure that attempts to insert data.
    - Include an `EXCEPTION` block to catch potential errors (e.g., primary key violation).
    - If an error occurs, `ROLLBACK` all changes made within the transaction.
4. **Create Procedure with Savepoint:**
    - Implement a procedure that performs several DML operations.
    - Use `SAVEPOINT` to mark a point in the transaction.
    - Demonstrate rolling back to the savepoint if a specific condition or error occurs.
5. **Test Procedures:** Call the procedures and verify the state of the database before and after their execution.

Source Code:

```
-- PL/SQL Procedures for Transaction Control (Oracle syntax)

-- 1. Procedure to add a new course and faculty, committing on success
CREATE OR REPLACE PROCEDURE AddCourseAndFaculty (
    p_course_id   IN VARCHAR2,
    p_course_name IN VARCHAR2,
    p_credits     IN NUMBER,
    p_dept_id     IN VARCHAR2,
    p_faculty_id  IN VARCHAR2,
    p_first_name  IN VARCHAR2,
    p_last_name   IN VARCHAR2,
    p_email       IN VARCHAR2,
    p_hire_date   IN DATE
)
IS
BEGIN
    INSERT INTO Course (CourseID, CourseName, Credits, DeptID)
    VALUES (p_course_id, p_course_name, p_credits, p_dept_id);

    INSERT INTO Faculty (FacultyID, FirstName, LastName, Email, HireDate,
DeptID)
    VALUES (p_faculty_id, p_first_name, p_last_name, p_email, p_hire_date,
p_dept_id);

    COMMIT; -- Commit both inserts if successful
    DBMS_OUTPUT.PUT_LINE('Course and Faculty added successfully and
committed.');
EXCEPTION
    WHEN OTHERS THEN
```

```
        ROLLBACK; -- Rollback both inserts if any error occurs
        DBMS_OUTPUT.PUT_LINE('Error adding Course and Faculty. Transaction
rolled back: ' || SQLERRM);
END;
/

-- Test AddCourseAndFaculty (Successful)
EXEC AddCourseAndFaculty('C008', 'Database Security', 3, 'CS', 'F005', 'Robert',
'Green', 'robert.g@example.com', '2023-01-10');

-- Test AddCourseAndFaculty (Rollback due to duplicate FacultyID)
-- This will cause a primary key violation for F005 if it already exists
EXEC AddCourseAndFaculty('C009', 'Data Mining', 3, 'CS', 'F005', 'Another',
'Faculty', 'another.f@example.com', '2023-02-15');


-- 2. Procedure demonstrating SAVEPOINT and partial rollback
CREATE OR REPLACE PROCEDURE UpdateStudentAndEnroll (
    p_student_id IN VARCHAR2,
    p_new_email  IN VARCHAR2,
    p_course_id  IN VARCHAR2
)
IS
BEGIN
    UPDATE Student
    SET Email = p_new_email
    WHERE StudentID = p_student_id;

    SAVEPOINT after_student_update;

    -- Attempt to enroll student in course
    INSERT INTO Enrollment (StudentID, CourseID)
    VALUES (p_student_id, p_course_id);

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Student updated and enrolled successfully.');
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN -- Specific error for unique constraint violation
        DBMS_OUTPUT.PUT_LINE('Student already enrolled in this course. Rolling
back enrollment only.');
        ROLLBACK TO SAVEPOINT after_student_update;
        COMMIT; -- Commit the student email update
    WHEN OTHERS THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred. Full transaction
rolled back: ' || SQLERRM);
END;
/

-- Test UpdateStudentAndEnroll (Successful)
EXEC UpdateStudentAndEnroll('S001', 'alice.new@example.com', 'C001'); --
Assuming S001 is not in C001

-- Test UpdateStudentAndEnroll (Rollback enrollment due to duplicate, but commit
student update)
-- Assuming S001 is already enrolled in C001
EXEC UpdateStudentAndEnroll('S001', 'alice.updated@example.com', 'C001');
```

Input: PL/SQL procedure definitions and execution commands.

Expected Output:

- Successful compilation messages for the procedures.
- Messages indicating successful commits or rollbacks.

- Verification of data changes (or lack thereof) in the database tables after procedure execution, confirming transaction control.

Lab 11: Develop test cases to demonstrate how timestamp-based protocols prevent conflicts and ensure serializability.

Title: Demonstrating Timestamp-Based Protocols for Concurrency Control

Aim: To understand and demonstrate how timestamp-based concurrency control protocols (e.g., basic timestamp ordering) prevent conflicts and ensure serializability in a multi-user database environment.

Procedure:

1. **Theoretical Understanding:**
    - Research timestamp-based concurrency control protocols.
    - Understand the concept of a timestamp for each transaction (assigned at transaction start).
    - Learn the rules for Read-Timestamp (R-TS) and Write-Timestamp (W-TS) for data items.
    - Understand how conflicts (read-write, write-read, write-write) are detected and handled (e.g., rolling back a transaction).
    - Explain how these rules ensure serializability (equivalent to some serial execution).
2. **Simulated Environment Setup:**
    - This lab is often best demonstrated conceptually or with a simplified simulation, as direct manipulation of internal timestamp mechanisms in commercial DBs is not usually exposed.
    - If using a database, set up a simple table with a few data items.
    - Simulate concurrent transactions by opening multiple database sessions.
3. **Develop Test Cases:**
    - **Scenario 1 (No Conflict):** Two transactions accessing different data items.
    - **Scenario 2 (Read-Write Conflict Prevention):**
        - T1 reads data item X.
        - T2 attempts to write data item X with a timestamp older than T1's read timestamp. (T2 should be rolled back).
    - **Scenario 3 (Write-Read Conflict Prevention):**
        - T1 writes data item X.
        - T2 attempts to read data item X with a timestamp older than T1's write timestamp. (T2 should be rolled back).
    - **Scenario 4 (Write-Write Conflict Prevention):**
        - T1 writes data item X.
        - T2 attempts to write data item X with a timestamp older than T1's write timestamp. (T2 should be rolled back).
4. **Execute and Observe:** Run the simulated transactions and observe which transactions are allowed to proceed and which are rolled back, demonstrating the conflict prevention mechanism.

Source Code: *This lab is primarily conceptual and would involve a high-level simulation or detailed explanation of transaction behavior rather than direct SQL code manipulating timestamps, as databases handle this internally. Below is a conceptual representation.*

```
-- Conceptual Simulation of Timestamp Protocol (Not actual SQL)

-- Assume a data item X with R_TS(X) and W_TS(X)
-- Assume Transaction T1 with TS(T1) = 10
```

```
-- Assume Transaction T2 with TS(T2) = 20

-- Initial state: X = 100, R_TS(X) = 0, W_TS(X) = 0

-- Test Case 1: T1 reads X, then T2 reads X (No conflict)
-- T1: READ(X)
--   If TS(T1) < W_TS(X) then ROLLBACK T1 (not the case here)
--   Else READ X, R_TS(X) = MAX(R_TS(X), TS(T1))
--   X = 100, R_TS(X) = 10, W_TS(X) = 0

-- T2: READ(X)
--   If TS(T2) < W_TS(X) then ROLLBACK T2 (not the case here)
--   Else READ X, R_TS(X) = MAX(R_TS(X), TS(T2))
--   X = 100, R_TS(X) = 20, W_TS(X) = 0

-- Test Case 2: Read-Write Conflict (T2 attempts to write X, but T1 read it
later)
-- Initial state: X = 100, R_TS(X) = 0, W_TS(X) = 0
-- T1: READ(X) (TS(T1) = 10)
--   R_TS(X) becomes 10

-- T2: WRITE(X) (TS(T2) = 5) -- T2 has an older timestamp than T1's read
--   If TS(T2) < R_TS(X) then ROLLBACK T2 (TRUE, 5 < 10, so T2 rolls back)
--   Else If TS(T2) < W_TS(X) then ROLLBACK T2
--   Else WRITE X, W_TS(X) = TS(T2)

-- Test Case 3: Write-Read Conflict (T2 attempts to read X, but T1 wrote it
later)
-- Initial state: X = 100, R_TS(X) = 0, W_TS(X) = 0
-- T1: WRITE(X) (TS(T1) = 10)
--   W_TS(X) becomes 10

-- T2: READ(X) (TS(T2) = 5) -- T2 has an older timestamp than T1's write
--   If TS(T2) < W_TS(X) then ROLLBACK T2 (TRUE, 5 < 10, so T2 rolls back)
--   Else READ X, R_TS(X) = MAX(R_TS(X), TS(T2))
```

Input:

- Conceptual scenarios for concurrent transactions.
- Simulated execution steps showing timestamp comparisons and outcomes.

Expected Output:

- Clear explanation of how timestamps are assigned and used.
- Demonstration of conflict detection and resolution (e.g., which transaction is rolled back).
- Proof that the protocol ensures serializability by preventing "dirty reads," "unrepeatable reads," and "phantom reads."

Lab 12: Case Study: Analyze different types of failures such as transaction failures, system crashes, and disk failures.

Title: Case Study: Analyzing Database Failures and Recovery Mechanisms

Aim: To understand and analyze various types of database failures (transaction, system, disk) and the corresponding recovery mechanisms employed by database management systems to ensure data durability and consistency.

Procedure:

1. **Theoretical Study:**
    - **Transaction Failures:** Understand causes (e.g., logical errors, deadlocks, integrity violations) and the role of `ROLLBACK`.
    - **System Crashes:** Understand causes (e.g., power failure, OS crash, software bug) and the need for recovery using logs (redo/undo).
    - **Disk Failures:** Understand causes (e.g., head crash, bad sectors) and the importance of backups and replication.
    - **ACID Properties:** Relate failure analysis to the ACID properties, particularly durability.
    - **Recovery Components:** Learn about the database log, checkpoints, and recovery algorithms (e.g., ARIES).
2. **Case Study Analysis:**
    - Choose a hypothetical or real-world scenario for each failure type.
    - **Transaction Failure:** Describe a transaction that fails due to a constraint violation. Explain how the DBMS rolls back the incomplete transaction.
    - **System Crash:** Describe a scenario where the database server crashes mid-transaction. Explain the recovery process: identifying committed transactions (redo) and uncommitted transactions (undo) using the log.
    - **Disk Failure:** Describe a scenario where a data disk is corrupted. Explain how the database is restored from a backup and then recovered using the transaction log to bring it to the latest consistent state.
3. **Discussion:** Discuss the impact of each failure type on data integrity and availability, and the effectiveness of different recovery strategies.

Source Code: *This lab is a case study and analysis, not a coding exercise. No direct source code is involved. The "source code" here refers to the descriptive analysis.*

Input:

- Descriptions of various failure scenarios.
- Knowledge of database recovery concepts (logs, checkpoints, ARIES).

Expected Output: A detailed report or presentation analyzing each failure type, including:

- **Failure Type:** Definition and common causes.
- **Impact:** Consequences on data and system.
- **Recovery Mechanism:** How the DBMS handles the failure (e.g., `ROLLBACK`, log-based recovery, backup restoration).
- **Example Scenario:** A concrete example demonstrating the failure and recovery.
- **Discussion:** How the recovery mechanism ensures ACID properties (especially durability).

Lab 13: Parallel Database

Title: Understanding and Implementing Concepts of Parallel Databases

Aim: To explore the architecture and principles of parallel database systems, understanding how they enhance performance by executing operations concurrently across multiple processors and disks.

Procedure:

1. **Theoretical Study:**
    - **Motivation:** Understand why parallel databases are needed (large data volumes, complex queries).
    - **Architectures:** Study different parallel database architectures: Shared-Memory, Shared-Disk, Shared-Nothing. Focus on the advantages and disadvantages of each.
    - **Parallel Operations:** Learn about parallel query processing techniques:
        - **Parallel Scan:** Reading data from multiple disks concurrently.
        - **Parallel Sort:** Sorting data across multiple processors.
        - **Parallel Join:** Different strategies for parallelizing join operations (e.g., hash join, sort-merge join).
        - **Parallel Aggregation:** Aggregating data concurrently.
    - **Data Partitioning (Sharding):** Understand different data partitioning schemes (e.g., hash partitioning, range partitioning, round-robin partitioning) and their impact on parallel query execution.
2. **Conceptual/Simulated Demonstration:**
    - **Query Optimization:** Discuss how a query optimizer might generate a parallel execution plan.
    - **Data Distribution:** Illustrate how data is distributed across nodes in a Shared-Nothing architecture.
    - **Performance Comparison:** Conceptually compare the execution time of a large query on a serial vs. parallel system.
3. **Practical Application (if environment allows):**
    - If access to a parallel database system (e.g., Greenplum, Teradata, or a distributed setup of PostgreSQL with Citus) is available, execute a large query and observe its parallel execution plan.
    - Demonstrate the impact of data partitioning on query performance.

Source Code: *This lab is primarily theoretical and conceptual. Actual parallel database implementation requires specialized environments. The "source code" would be illustrative SQL demonstrating concepts.*

```
-- Conceptual SQL for Parallel Query (actual execution depends on DB system)

-- Assume a very large 'Sales' table in a parallel database
-- with 'Region' as a partitioning key.

-- Example of a query that could be parallelized:
SELECT Region, SUM(SalesAmount) AS TotalSales
FROM LargeSalesTable
WHERE SaleDate BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY Region;

-- Explanation of how this query might be parallelized:
-- 1. Parallel Scan: Each node scans its partition of LargeSalesTable.
```

```
-- 2. Parallel Filter: Each node filters data for the specified date range.
-- 3. Parallel Grouping/Aggregation:
--    a. Local aggregation: Each node calculates SUM(SalesAmount) for its local
'Region' data.
--    b. Data redistribution: If 'Region' is not the partitioning key,
intermediate results
--       might be shuffled to nodes responsible for specific regions.
--    c. Global aggregation: Final aggregation is performed on the shuffled
data.

-- Example of creating a partitioned table (conceptual, syntax varies widely)
-- CREATE TABLE Orders (
--     OrderID INT,
--     OrderDate DATE,
--     CustomerID INT,
--     Amount DECIMAL(10, 2)
-- )
-- PARTITION BY RANGE (YEAR(OrderDate)) (
--     PARTITION p2022 VALUES LESS THAN (2023),
--     PARTITION p2023 VALUES LESS THAN (2024),
--     PARTITION p2024 VALUES LESS THAN (2025),
--     PARTITION p_future VALUES LESS THAN MAXVALUE
-- );
```

Input:

- Theoretical concepts of parallel database architectures and operations.
- Illustrative SQL queries for large datasets.

Expected Output:

- Detailed explanation of parallel database architectures (Shared-Memory, Shared-Disk, Shared-Nothing) with their pros and cons.
- Description of how common database operations (scan, sort, join, aggregation) are parallelized.
- Conceptual diagrams illustrating data partitioning and parallel query execution plans.
- Discussion of performance benefits and challenges of parallel databases.

Lab 14: Case Study: distributed Database

Title: Case Study: Analyzing Distributed Database Systems

Aim: To understand the concepts, challenges, and solutions associated with distributed database systems, focusing on data distribution, query processing, transaction management, and consistency models in a distributed environment.

Procedure:

1. **Theoretical Study:**
    - **Definition:** What is a distributed database system (DDBS)?
    - **Reasons for Distribution:** Scalability, availability, local autonomy, cost.
    - **Architectures:** Homogeneous vs. Heterogeneous, Client-Server, Peer-to-Peer.
    - **Transparency:** Data, query, transaction, schema, network, replication transparency.
    - **Data Distribution:**
        - **Fragmentation:** Horizontal, Vertical, Mixed.
        - **Replication:** Full, Partial.
    - **Distributed Query Processing:** Optimization challenges (e.g., semi-join).
    - **Distributed Transaction Management:** Two-Phase Commit (2PC), Three-Phase Commit (3PC).
    - **Concurrency Control:** Distributed locking, distributed timestamping.
    - **Consistency Models:** ACID vs. BASE, Eventual Consistency.
2. **Case Study Analysis:**
    - Choose a scenario requiring a distributed database (e.g., a global e-commerce platform, a multi-national bank).
    - **Scenario Description:** Detail the business requirements and geographical distribution.
    - **Design Choices:**
        - How would data be fragmented and replicated? (e.g., customer data fragmented by region, product catalog replicated).
        - Which consistency model would be appropriate for different data types?
        - How would distributed queries (e.g., "find all orders from Europe and Asia") be handled?
        - How would distributed transactions (e.g., "transfer money between accounts in different regions") be managed using 2PC?
    - **Challenges:** Discuss potential issues like network latency, partial failures, distributed deadlocks, and maintaining consistency.
    - **Solutions:** Propose solutions for the identified challenges.
3. **Discussion:** Compare and contrast distributed databases with centralized and parallel databases.

Source Code: *This lab is a case study and analysis, not a coding exercise. No direct source code is involved. The "source code" here refers to the descriptive analysis.*

Input:

- Theoretical concepts of distributed databases.
- A real-world or hypothetical scenario for a distributed system.

Expected Output: A comprehensive report or presentation analyzing a chosen distributed database case study, including:

- **Introduction to Distributed Databases:** Definition, advantages, and challenges.
- **Case Study Description:** Detailed scenario (e.g., a global retail chain).
- **Distributed Design:** Proposed fragmentation, replication, and consistency strategies.
- **Distributed Query/Transaction Processing:** Explanation of how these are handled in the chosen design.
- **Challenges and Solutions:** Identification of specific problems (e.g., network partitions, data inconsistencies) and proposed mitigation strategies.
- **Conclusion:** Summary of findings and comparison with other database paradigms.

Lab 15: Creating database employee in MongoDB

Title: Creating an Employee Database in MongoDB

Aim: To gain hands-on experience with NoSQL databases by creating and managing an employee database using MongoDB, focusing on document-oriented data modeling and basic CRUD operations.

Procedure:

1. **MongoDB Setup:**
    - Install MongoDB Community Server.
    - Install MongoDB Shell (mongosh) or a GUI client (e.g., MongoDB Compass).
2. **Connect to MongoDB:** Connect to the MongoDB instance using the shell or client.
3. **Database and Collection Creation:**
    - Switch to or create a new database (e.g., `employeeDB`).
    - Create a collection (e.g., `employees`).
4. **Document-Oriented Data Modeling:**
    - Design a schema-less document structure for an employee, including fields like `_id`, `firstName`, `lastName`, `email`, `department`, `position`, `salary`, `hireDate`, and potentially nested documents for `contact` or `address`.
    - Understand the flexibility of embedding documents and arrays.
5. **CRUD Operations:**
    - **Create (Insert):** Insert single and multiple employee documents into the `employees` collection.
    - **Read (Find):** Query documents using `find()`, with various criteria, projections, and sorting.
    - **Update:** Modify existing employee documents using `updateOne()`, `updateMany()`, and update operators (`$set`, `$inc`, `$push`, etc.).
    - **Delete:** Remove documents using `deleteOne()` and `deleteMany()`.
6. **Indexing:** Create indexes on frequently queried fields to improve performance.

Source Code:

```
// MongoDB Shell (mongosh) commands

// 1. Switch to/Create Database
use employeeDB;

// 2. Insert a single employee document
db.employees.insertOne({
    firstName: "John",
    lastName: "Doe",
    email: "john.doe@example.com",
    department: "IT",
    position: "Software Engineer",
    salary: 75000,
    hireDate: ISODate("2020-01-15"),
    contact: {
        phone: "123-456-7890",
        address: {
            street: "123 Main St",
            city: "Anytown",
            state: "CA",
            zip: "90210"
```

```
        }
    },
    skills: ["JavaScript", "Node.js", "MongoDB"]
});

// 3. Insert multiple employee documents
db.employees.insertMany([
    {
        firstName: "Jane",
        lastName: "Smith",
        email: "jane.smith@example.com",
        department: "HR",
        position: "HR Manager",
        salary: 80000,
        hireDate: ISODate("2018-03-01"),
        contact: { phone: "987-654-3210" },
        skills: ["Recruitment", "Employee Relations"]
    },
    {
        firstName: "Peter",
        lastName: "Jones",
        email: "peter.j@example.com",
        department: "IT",
        position: "Database Administrator",
        salary: 90000,
        hireDate: ISODate("2019-07-20"),
        skills: ["MongoDB", "SQL", "Linux"]
    }
]);

// 4. Find all employees
db.employees.find({});

// 5. Find employees in the 'IT' department
db.employees.find({ department: "IT" });

// 6. Find employees with salary greater than 80000
db.employees.find({ salary: { $gt: 80000 } });

// 7. Find employees with 'MongoDB' skill
db.employees.find({ skills: "MongoDB" });

// 8. Update John Doe's salary
db.employees.updateOne(
    { firstName: "John", lastName: "Doe" },
    { $set: { salary: 80000, position: "Senior Software Engineer" } }
);

// 9. Add a new skill to Jane Smith
db.employees.updateOne(
    { firstName: "Jane", lastName: "Smith" },
    { $push: { skills: "Training" } }
);

// 10. Delete an employee by email
db.employees.deleteOne({ email: "jane.smith@example.com" });

// 11. Create an index on the 'email' field for faster lookups
db.employees.createIndex({ email: 1 });
```

Input: MongoDB Shell commands for database operations.

Expected Output:

- Confirmation messages for successful insertions, updates, and deletions.

- JSON documents displayed in the shell when `find()` queries are executed, showing the retrieved data.
- Output confirming index creation.