**SRM Institute of Science and Technology**

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

**Department of Computer Applications**

**Circular – 2023-24**

**B.Sc. CS 3rd Sem**

# DATABASE MANAGEMENT SYSTEM (USA23302)- Lab Manual

## Lab 1: Data Definition Language Commands on sample exercise

### Aim

To understand and implement Data Definition Language (DDL) commands in SQL to create, modify, and delete database objects like tables, views, and indexes.

### Procedure

1. **Connect to the Database:** Establish a connection to your SQL database (e.g., Oracle, MySQL, PostgreSQL).
2. **Create a Table:** Use the `CREATE TABLE` command to define a new table with appropriate columns, data types, and constraints (e.g., `PRIMARY KEY`, `NOT NULL`, `UNIQUE`).
3. **Alter a Table:** Use the `ALTER TABLE` command to add new columns, modify existing column definitions, or drop columns from an existing table.
4. **Drop a Table:** Use the `DROP TABLE` command to delete an entire table from the database.
5. **Rename a Table:** Use the `RENAME TABLE` command (or `ALTER TABLE RENAME TO` depending on the SQL dialect) to change the name of an existing table.
6. **Truncate a Table:** Use the `TRUNCATE TABLE` command to remove all rows from a table, but keep the table structure intact.

### Source Code

```
-- 1. Create a new table named 'Students'
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Age INT,
    Major VARCHAR(100)
);

-- 2. Add a new column 'Email' to the 'Students' table
ALTER TABLE Students
ADD Email VARCHAR(100);

-- 3. Modify the data type of the 'Major' column
ALTER TABLE Students
MODIFY Major VARCHAR(150); -- Syntax might vary (e.g., ALTER COLUMN in SQL
Server)

-- 4. Rename the 'Students' table to 'UniversityStudents'
ALTER TABLE Students
```

```
RENAME TO UniversityStudents; -- Syntax might vary (e.g., RENAME TABLE
Students TO UniversityStudents;)

-- 5. Create another table 'Courses'
CREATE TABLE Courses (
    CourseID VARCHAR(10) PRIMARY KEY,
    CourseName VARCHAR(100) NOT NULL,
    Credits INT
);

-- 6. Drop the 'Courses' table
DROP TABLE Courses;

-- 7. Truncate the 'UniversityStudents' table (after inserting some data for
demonstration)
-- INSERT INTO UniversityStudents (StudentID, FirstName, LastName, Age,
Major, Email) VALUES (1, 'John', 'Doe', 20, 'Computer Science',
'john.doe@example.com');
-- TRUNCATE TABLE UniversityStudents;
```

## Input

No specific input is required for DDL commands as they modify the database schema directly.

## Expected Output

The successful execution of the DDL commands will result in:

- Creation of the `Students` table.
- Addition of the `Email` column to `Students`.
- Modification of the `Major` column's data type.
- Renaming of the `Students` table to `UniversityStudents`.
- Deletion of the `Courses` table.
- Emptying of all rows from the `UniversityStudents` table (if data was inserted and then truncated).
- Confirmation messages from the SQL client indicating successful command execution (e.g., "Table created.", "Table altered.", "Table dropped.").

# Lab 2: SQL Data Manipulation Language Commands

## Aim

To learn and practice Data Manipulation Language (DML) commands in SQL to insert, update, delete, and retrieve data from database tables.

## Procedure

1. **Connect to the Database:** Establish a connection to your SQL database.
2. **Insert Data:** Use the `INSERT INTO` command to add new rows of data into a table.
3. **Update Data:** Use the `UPDATE` command to modify existing data in one or more rows of a table based on specified conditions.
4. **Delete Data:** Use the `DELETE FROM` command to remove one or more rows from a table based on specified conditions.
5. **Retrieve Data:** Use the `SELECT` command to fetch data from one or more tables.

## Source Code

```
-- Assuming the 'UniversityStudents' table from Lab 1 exists and is empty
after truncation.
-- If not, create it first:
-- CREATE TABLE UniversityStudents (
--     StudentID INT PRIMARY KEY,
--     FirstName VARCHAR(50) NOT NULL,
--     LastName VARCHAR(50) NOT NULL,
--     Age INT,
--     Major VARCHAR(150),
--     Email VARCHAR(100)
-- );

-- 1. Insert data into the 'UniversityStudents' table
INSERT INTO UniversityStudents (StudentID, FirstName, LastName, Age, Major,
Email) VALUES
(101, 'Alice', 'Smith', 21, 'Computer Science', 'alice.s@example.com'),
(102, 'Bob', 'Johnson', 22, 'Electrical Engineering', 'bob.j@example.com'),
(103, 'Charlie', 'Brown', 20, 'Mathematics', 'charlie.b@example.com'),
(104, 'Diana', 'Prince', 23, 'Computer Science', 'diana.p@example.com');

-- 2. Update the Major of Alice Smith
UPDATE UniversityStudents
SET Major = 'Software Engineering'
WHERE StudentID = 101;

-- 3. Update the Age of all students in Computer Science
UPDATE UniversityStudents
SET Age = Age + 1
WHERE Major = 'Computer Science';

-- 4. Delete the student with StudentID 103
DELETE FROM UniversityStudents
WHERE StudentID = 103;

-- 5. Select all data from the 'UniversityStudents' table
SELECT * FROM UniversityStudents;

-- 6. Select FirstName and Major for students older than 22
SELECT FirstName, Major
FROM UniversityStudents
WHERE Age > 22;
```

## Input

The `INSERT` statements provide the input data. For `UPDATE` and `DELETE`, the `WHERE` clauses specify which rows are affected.

## Expected Output

After executing the DML commands, the `SELECT` statements should produce the following (or similar, depending on the exact order of operations and initial state):

```
-- After INSERTs, UPDATEs, and DELETEs, and then SELECT * FROM
UniversityStudents;
StudentID | FirstName | LastName | Age | Major                | Email
----------|-----------|----------|-----|----------------------|--------------
---------
101       | Alice     | Smith    | 22  | Software Engineering |
alice.s@example.com
102       | Bob       | Johnson  | 22  | Electrical Engineering |
bob.j@example.com
104       | Diana     | Prince   | 24  | Computer Science     |
diana.p@example.com

-- After SELECT FirstName, Major FROM UniversityStudents WHERE Age > 22;
FirstName | Major
----------|------------------
Diana     | Computer Science
```

# Lab 3: SQL Data Control Language Commands and Transaction control commands to the sample exercises

## Aim

To understand and implement Data Control Language (DCL) commands for managing database permissions and Transaction Control Language (TCL) commands for managing database transactions.

## Procedure

1. **Connect as Administrator/Privileged User:** Log in to the database with a user account that has sufficient privileges to grant and revoke permissions.
2. **Create a New User:** Use `CREATE USER` (or similar, depending on the database) to create a new database user.
3. **Grant Permissions:** Use the `GRANT` command to assign specific privileges (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`) on tables or other database objects to a user or role.
4. **Revoke Permissions:** Use the `REVOKE` command to remove previously granted privileges.
5. **Start a Transaction:** Use `START TRANSACTION` or `BEGIN TRANSACTION` to mark the beginning of a transaction.
6. **Perform DML Operations:** Execute `INSERT`, `UPDATE`, or `DELETE` statements within the transaction.
7. **Commit Transaction:** Use `COMMIT` to save all changes made within the transaction permanently to the database.
8. **Rollback Transaction:** Use `ROLLBACK` to undo all changes made within the transaction, restoring the database to its state before the transaction began.
9. **Savepoint (Optional):** Use `SAVEPOINT` to set a point within a transaction to which you can later roll back.

## Source Code

```
-- Assuming a user 'test_user' is to be created and managed.
-- Syntax for user creation and granting privileges can vary significantly
between databases (e.g., Oracle, MySQL).

-- DCL Commands (Example for MySQL/PostgreSQL-like syntax)
-- 1. Create a new user (requires administrator privileges)
-- CREATE USER 'test_user'@'localhost' IDENTIFIED BY 'password'; -- MySQL
-- CREATE USER test_user WITH PASSWORD 'password'; -- PostgreSQL

-- 2. Grant SELECT and INSERT privileges on UniversityStudents table to
'test_user'
GRANT SELECT, INSERT ON UniversityStudents TO 'test_user'@'localhost'; --
MySQL
-- GRANT SELECT, INSERT ON UniversityStudents TO test_user; -- PostgreSQL

-- 3. Grant all privileges on UniversityStudents table to 'test_user'
-- GRANT ALL PRIVILEGES ON UniversityStudents TO 'test_user'@'localhost'; --
MySQL
-- GRANT ALL PRIVILEGES ON UniversityStudents TO test_user; -- PostgreSQL

-- 4. Revoke INSERT privilege on UniversityStudents table from 'test_user'
REVOKE INSERT ON UniversityStudents TO 'test_user'@'localhost'; -- MySQL
-- REVOKE INSERT ON UniversityStudents FROM test_user; -- PostgreSQL

-- TCL Commands (Example)
-- Assuming we are operating on the 'UniversityStudents' table
```

```
-- Scenario 1: Successful Transaction
START TRANSACTION;

INSERT INTO UniversityStudents (StudentID, FirstName, LastName, Age, Major,
Email) VALUES
(105, 'Frank', 'Green', 21, 'Physics', 'frank.g@example.com');

UPDATE UniversityStudents
SET Age = 22
WHERE StudentID = 105;

COMMIT; -- Saves the changes

-- Scenario 2: Transaction with Rollback
START TRANSACTION;

INSERT INTO UniversityStudents (StudentID, FirstName, LastName, Age, Major,
Email) VALUES
(106, 'Grace', 'Hall', 20, 'Chemistry', 'grace.h@example.com');

DELETE FROM UniversityStudents
WHERE StudentID = 101; -- Intentionally deleting Alice Smith

ROLLBACK; -- Undoes the INSERT and DELETE operations

-- Scenario 3: Transaction with Savepoint (Optional)
START TRANSACTION;

INSERT INTO UniversityStudents (StudentID, FirstName, LastName, Age, Major,
Email) VALUES
(107, 'Henry', 'King', 22, 'Biology', 'henry.k@example.com');

SAVEPOINT before_update; -- Set a savepoint

UPDATE UniversityStudents
SET Major = 'Environmental Science'
WHERE StudentID = 107;

-- ROLLBACK TO before_update; -- If you want to undo only the update
COMMIT;
```

## Input

- **DCL:** Usernames and specific privileges to be granted or revoked.
- **TCL:** DML statements (`INSERT`, `UPDATE`, `DELETE`) executed within the transaction blocks.

## Expected Output

- **DCL:** Confirmation messages indicating successful granting or revoking of privileges. When `test_user` attempts operations without granted permissions, they should receive "permission denied" errors.
- **TCL:**
  - **Successful Transaction (Scenario 1):** The new student (Frank Green) will be permanently added to the `UniversityStudents` table, and their age will be updated.
  - **Transaction with Rollback (Scenario 2):** The new student (Grace Hall) will *not* be added, and Alice Smith will *not* be deleted from the `UniversityStudents` table, as the `ROLLBACK` command will undo all changes within that transaction.

- **Transaction with Savepoint (Scenario 3):** Henry King will be added, and their major will be updated if `COMMIT` is executed. If `ROLLBACK TO before_update` was used, only the update would be undone, but the insert would remain (until a full rollback or commit).

# Lab 4: Inbuilt functions in SQL on sample Exercise

## Aim

To explore and apply various inbuilt (built-in) functions in SQL for data manipulation, aggregation, and formatting.

## Procedure

1. **Connect to the Database:** Establish a connection to your SQL database.
2. **String Functions:** Use functions like `UPPER()`, `LOWER()`, `LENGTH()`, `SUBSTRING()`, `CONCAT()` to manipulate string data.
3. **Numeric Functions:** Use functions like `ROUND()`, `ABS()`, `SQRT()`, `MOD()` for mathematical operations.
4. **Date/Time Functions:** Use functions like `CURRENT_DATE()`, `NOW()`, `DATE_FORMAT()`, `DATEDIFF()` to work with date and time values.
5. **Aggregate Functions:** Use functions like `COUNT()`, `SUM()`, `AVG()`, `MIN()`, `MAX()` to perform calculations on sets of rows.
6. **Conversion Functions:** Use functions like `CAST()` or `CONVERT()` to change data types.

## Source Code

```
-- Assuming the 'UniversityStudents' table from previous labs
-- And let's add a 'EnrollmentDate' column for date functions
ALTER TABLE UniversityStudents
ADD EnrollmentDate DATE;

UPDATE UniversityStudents
SET EnrollmentDate = '2022-09-01' WHERE StudentID = 101;
UPDATE UniversityStudents
SET EnrollmentDate = '2021-08-15' WHERE StudentID = 102;
UPDATE UniversityStudents
SET EnrollmentDate = '2023-01-20' WHERE StudentID = 104;
UPDATE UniversityStudents
SET EnrollmentDate = '2022-10-05' WHERE StudentID = 105; -- If Frank Green
was committed

-- 1. String Functions
SELECT
    FirstName,
    UPPER(LastName) AS UpperLastName,
    LENGTH(Major) AS MajorLength,
    SUBSTRING(Email, 1, INSTR(Email, '@') - 1) AS EmailUsername, -- INSTR for
MySQL/Oracle, CHARINDEX for SQL Server
    CONCAT(FirstName, ' ', LastName) AS FullName
FROM UniversityStudents;

-- 2. Numeric Functions
-- Let's assume we have a 'Grades' table for this
-- CREATE TABLE Grades (StudentID INT, Subject VARCHAR(50), Score
DECIMAL(5,2));
-- INSERT INTO Grades VALUES (101, 'DBMS', 85.5), (101, 'OS', 90.0), (102,
'DBMS', 78.2), (104, 'OS', 92.1);

-- SELECT StudentID, ROUND(AVG(Score), 2) AS AverageScore FROM Grades GROUP
BY StudentID;
-- SELECT ABS(-100) AS AbsoluteValue;

-- 3. Date/Time Functions
SELECT
    FirstName,
```

```
    EnrollmentDate,
    CURRENT_DATE() AS CurrentDate, -- Or GETDATE() in SQL Server, SYSDATE in
Oracle
    DATEDIFF(CURRENT_DATE(), EnrollmentDate) AS DaysSinceEnrollment --
DATEDIFF for MySQL, different syntax for others
FROM UniversityStudents;

-- 4. Aggregate Functions
SELECT
    COUNT(*) AS TotalStudents,
    AVG(Age) AS AverageAge,
    SUM(CASE WHEN Major = 'Computer Science' THEN 1 ELSE 0 END) AS
CSStudentsCount,
    MIN(Age) AS YoungestStudentAge,
    MAX(Age) AS OldestStudentAge
FROM UniversityStudents;

-- 5. Conversion Functions
SELECT
    CAST(StudentID AS CHAR(10)) AS StudentID_as_Char,
    'Age: ' || CAST(Age AS VARCHAR(3)) AS Age_as_String -- || for
concatenation in Oracle/PostgreSQL
FROM UniversityStudents;
```

## Input

The data within the `UniversityStudents` table (and `Grades` table if created) serves as input for these functions.

## Expected Output

The output will be new columns or aggregated values derived from the existing data using the specified functions. For example:

```
-- Sample output for String Functions
FirstName | UpperLastName | MajorLength | EmailUsername | FullName
----------|---------------|-------------|---------------|------------------
Alice     | SMITH         | 20          | alice.s       | Alice Smith
Bob       | JOHNSON       | 22          | bob.j         | Bob Johnson
Diana     | PRINCE        | 16          | diana.p       | Diana Prince
Frank     | GREEN         | 7           | frank.g       | Frank Green

-- Sample output for Aggregate Functions
TotalStudents | AverageAge | CSStudentsCount | YoungestStudentAge |
OldestStudentAge
--------------|------------|-----------------|--------------------|----------
-------
4             | 22.25      | 1               | 21                 | 24
```

(Note: Actual values for Age and CSStudentsCount depend on the exact state of your `UniversityStudents` table after previous labs).

# Lab 5: SQL Queries and Set operation SQL

## Aim

To master the art of writing complex SQL queries using various clauses (`WHERE`, `ORDER BY`, `GROUP BY`, `HAVING`) and to understand and apply set operations (`UNION`, `INTERSECT`, `EXCEPT`/`MINUS`).

## Procedure

1. **Connect to the Database:** Establish a connection to your SQL database.
2. **Basic Queries:** Practice `SELECT` statements with `WHERE` clauses for filtering.
3. **Ordering Results:** Use `ORDER BY` to sort query results in ascending or descending order.
4. **Grouping Data:** Use `GROUP BY` with aggregate functions to group rows that have the same values in specified columns into a summary row.
5. **Filtering Groups:** Use `HAVING` to filter the results of `GROUP BY` based on aggregate conditions.
6. **Set Operations:**
    - `UNION` / `UNION ALL`: Combine the result sets of two or more `SELECT` statements (removes duplicates for `UNION`, keeps all for `UNION ALL`).
    - `INTERSECT`: Return only the rows that are common to both result sets.
    - `EXCEPT` / `MINUS`: Return rows from the first result set that are not present in the second.

## Source Code

```
-- Assuming UniversityStudents table exists
-- Let's create another table for demonstration of set operations
CREATE TABLE Alumni (
    AlumniID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    GraduationYear INT
);

INSERT INTO Alumni (AlumniID, FirstName, LastName, GraduationYear) VALUES
(201, 'Alice', 'Smith', 2024), -- Alice from UniversityStudents
(202, 'Peter', 'Jones', 2023),
(102, 'Bob', 'Johnson', 2024); -- Bob from UniversityStudents

-- SQL Queries
-- 1. Select students majoring in Computer Science, ordered by age
SELECT StudentID, FirstName, LastName, Age, Major
FROM UniversityStudents
WHERE Major = 'Computer Science'
ORDER BY Age DESC;

-- 2. Count students per major
SELECT Major, COUNT(StudentID) AS NumberOfStudents
FROM UniversityStudents
GROUP BY Major;

-- 3. Find majors with more than 1 student
SELECT Major, COUNT(StudentID) AS NumberOfStudents
FROM UniversityStudents
GROUP BY Major
HAVING COUNT(StudentID) > 1;

-- Set Operations
```

```
-- 1. UNION: Combine names from UniversityStudents and Alumni (removes
duplicates)
SELECT FirstName, LastName FROM UniversityStudents
UNION
SELECT FirstName, LastName FROM Alumni;

-- 2. UNION ALL: Combine names from UniversityStudents and Alumni (keeps all,
including duplicates)
SELECT FirstName, LastName FROM UniversityStudents
UNION ALL
SELECT FirstName, LastName FROM Alumni;

-- 3. INTERSECT: Find names that appear in both UniversityStudents and Alumni
-- Note: INTERSECT is not supported in MySQL. Use JOINs or subqueries for
equivalent.
-- For databases supporting INTERSECT (e.g., PostgreSQL, Oracle, SQL Server):
SELECT FirstName, LastName FROM UniversityStudents
INTERSECT
SELECT FirstName, LastName FROM Alumni;

-- Equivalent for MySQL using JOIN:
SELECT DISTINCT S.FirstName, S.LastName
FROM UniversityStudents S
INNER JOIN Alumni A ON S.FirstName = A.FirstName AND S.LastName = A.LastName;

-- 4. EXCEPT / MINUS: Find names in UniversityStudents that are not in Alumni
-- Note: EXCEPT is used in PostgreSQL/SQL Server, MINUS in Oracle. Not
directly in MySQL.
-- For databases supporting EXCEPT/MINUS:
SELECT FirstName, LastName FROM UniversityStudents
EXCEPT
SELECT FirstName, LastName FROM Alumni;

-- Equivalent for MySQL using LEFT JOIN and WHERE IS NULL:
SELECT S.FirstName, S.LastName
FROM UniversityStudents S
LEFT JOIN Alumni A ON S.FirstName = A.FirstName AND S.LastName = A.LastName
WHERE A.AlumniID IS NULL;
```

## Input

The data in the `UniversityStudents` and `Alumni` tables.

## Expected Output

- **SQL Queries:** Filtered, sorted, or aggregated results based on the query criteria.
  - Example for "majors with more than 1 student":
  - `Major | NumberOfStudents`
  - `---|---`
  - `Computer Science | 2`

- **Set Operations:**
  - **UNION:** A combined list of unique first and last names from both tables.
  - **UNION ALL:** A combined list of all first and last names from both tables, including duplicates.
  - **INTERSECT:** A list containing 'Alice Smith' and 'Bob Johnson' (if they exist in both).
  - **EXCEPT/MINUS:** A list of students who are currently in `UniversityStudents` but not yet in `Alumni` (e.g., Diana Prince, Frank Green).

# Lab 6: Nested Queries on sample exercise * Construction of Relational Table from the ER Diagram

## Aim

To understand and implement nested queries (subqueries) in SQL, and to practice constructing relational tables based on an Entity-Relationship (ER) Diagram.

## Procedure

1. **Understand Nested Queries:** Learn how a subquery (inner query) executes first and its result is used by the outer query.
2. **Types of Subqueries:** Practice using subqueries in `WHERE` clause (`IN`, `EXISTS`, comparison operators), `FROM` clause (derived tables), and `SELECT` clause (scalar subqueries).
3. **ER Diagram to Relational Model:** Given an ER Diagram, identify entities, attributes, relationships, primary keys, and foreign keys. Translate these components into a set of relational tables with appropriate columns, data types, and constraints.

## Source Code

```
-- Assuming UniversityStudents table and let's create a new table
'Departments'
CREATE TABLE Departments (
    DeptID VARCHAR(10) PRIMARY KEY,
    DeptName VARCHAR(100) NOT NULL,
    HeadOfDept VARCHAR(100)
);

INSERT INTO Departments (DeptID, DeptName, HeadOfDept) VALUES
('CS', 'Computer Science', 'Dr. Sharma'),
('EE', 'Electrical Engineering', 'Dr. Khan'),
('MA', 'Mathematics', 'Dr. Lee');

-- Update UniversityStudents to link to Departments
ALTER TABLE UniversityStudents
ADD DeptID VARCHAR(10);

UPDATE UniversityStudents SET DeptID = 'CS' WHERE Major = 'Computer Science'
OR Major = 'Software Engineering';
UPDATE UniversityStudents SET DeptID = 'EE' WHERE Major = 'Electrical
Engineering';
UPDATE UniversityStudents SET DeptID = 'MA' WHERE Major = 'Mathematics';
UPDATE UniversityStudents SET DeptID = 'PH' WHERE Major = 'Physics'; --
Assuming Physics is a new department

-- Nested Queries

-- 1. Subquery in WHERE clause (using IN): Find students in departments
located in 'Building A' (hypothetical column)
-- Let's assume Departments table has a Building column
ALTER TABLE Departments ADD Building VARCHAR(50);
UPDATE Departments SET Building = 'Building A' WHERE DeptID = 'CS';
UPDATE Departments SET Building = 'Building B' WHERE DeptID = 'EE';
UPDATE Departments SET Building = 'Building A' WHERE DeptID = 'MA';

SELECT FirstName, LastName, Major
FROM UniversityStudents
WHERE DeptID IN (SELECT DeptID FROM Departments WHERE Building = 'Building
A');
```

```
-- 2. Subquery in WHERE clause (using EXISTS): Find departments that have at
least one student
SELECT DeptName
FROM Departments d
WHERE EXISTS (SELECT 1 FROM UniversityStudents s WHERE s.DeptID = d.DeptID);

-- 3. Scalar Subquery in SELECT clause: Get student's name and their
department's head
SELECT
    FirstName,
    LastName,
    (SELECT HeadOfDept FROM Departments WHERE DeptID = s.DeptID) AS
DepartmentHead
FROM UniversityStudents s;

-- 4. Derived Table (Subquery in FROM clause): Find average age of students
per department
SELECT d.DeptName, avg_age.AverageAge
FROM Departments d
JOIN (
    SELECT DeptID, AVG(Age) AS AverageAge
    FROM UniversityStudents
    GROUP BY DeptID
) AS avg_age ON d.DeptID = avg_age.DeptID;

-- Construction of Relational Table from ER Diagram (Conceptual Example)
/*
Assume an ER Diagram with:
- Entity: BOOK (ISBN, Title, PublicationYear, Price)
- Entity: AUTHOR (AuthorID, Name, BirthDate)
- Relationship: Writes (between BOOK and AUTHOR, Many-to-Many)
    - Attributes on relationship: RoyaltyPercentage

Relational Tables:

1. BOOK
    - ISBN (Primary Key, VARCHAR(20))
    - Title (VARCHAR(255))
    - PublicationYear (INT)
    - Price (DECIMAL(10,2))

2. AUTHOR
    - AuthorID (Primary Key, INT)
    - Name (VARCHAR(100))
    - BirthDate (DATE)

3. WRITES (Junction Table for Many-to-Many relationship)
    - ISBN (Foreign Key referencing BOOK.ISBN, VARCHAR(20))
    - AuthorID (Foreign Key referencing AUTHOR.AuthorID, INT)
    - RoyaltyPercentage (DECIMAL(5,2))
    - Primary Key: (ISBN, AuthorID)
*/

-- SQL DDL for the ER Diagram example:
CREATE TABLE BOOK (
    ISBN VARCHAR(20) PRIMARY KEY,
    Title VARCHAR(255) NOT NULL,
    PublicationYear INT,
    Price DECIMAL(10,2)
);

CREATE TABLE AUTHOR (
    AuthorID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    BirthDate DATE
```

```
);

CREATE TABLE WRITES (
    ISBN VARCHAR(20),
    AuthorID INT,
    RoyaltyPercentage DECIMAL(5,2),
    PRIMARY KEY (ISBN, AuthorID),
    FOREIGN KEY (ISBN) REFERENCES BOOK(ISBN),
    FOREIGN KEY (AuthorID) REFERENCES AUTHOR(AuthorID)
);
```

## Input

The data in `UniversityStudents` and `Departments` tables. For ER Diagram, the conceptual ER diagram itself.

## Expected Output

- **Nested Queries:**
  - Example for "students in departments located in 'Building A'":

```
FirstName | LastName | Major
----------|----------|--------------------
Alice     | Smith    | Software Engineering
Diana     | Prince   | Computer Science
Charlie   | Brown    | Mathematics
```

  - Example for "departments that have at least one student":

```
DeptName
--------------------
Computer Science
Electrical Engineering
Mathematics
Physics
```

  - Example for "student's name and their department's head":

```
FirstName | LastName | DepartmentHead
----------|----------|---------------
Alice     | Smith    | Dr. Sharma
Bob       | Johnson  | Dr. Khan
Diana     | Prince   | Dr. Sharma
Frank     | Green    | NULL (or corresponding for 'PH')
```

- **ER Diagram to Relational Model:** Successful creation of `BOOK`, `AUTHOR`, and `WRITES` tables with their defined primary and foreign key constraints.

# Lab 7: Join Queries on sample exercise. Demonstration for all Join Commands with SQL queries

## Aim

To understand and implement various types of JOIN operations in SQL to combine rows from two or more tables based on a related column between them.

## Procedure

1. **Connect to the Database:** Establish a connection to your SQL database.
2. **INNER JOIN:** Combine rows from two tables where there is a match in both tables.
3. **LEFT (OUTER) JOIN:** Return all rows from the left table, and the matching rows from the right table. If there is no match, NULLs are returned for the right side.
4. **RIGHT (OUTER) JOIN:** Return all rows from the right table, and the matching rows from the left table. If there is no match, NULLs are returned for the left side.
5. **FULL (OUTER) JOIN:** Return all rows when there is a match in one of the tables. Returns rows from both tables even if there are no matches (not supported in MySQL).
6. **CROSS JOIN:** Returns the Cartesian product of the two tables (every row from the first table combined with every row from the second table).
7. **SELF JOIN:** Joining a table to itself.

## Source Code

```
-- Assuming UniversityStudents and Departments tables from Lab 6
-- Let's ensure some data for demonstration, including mismatches for OUTER
JOINs

-- Data in UniversityStudents: (101, Alice, CS), (102, Bob, EE), (104, Diana,
CS), (105, Frank, PH)
-- Data in Departments: (CS, Computer Science), (EE, Electrical Engineering),
(MA, Mathematics)

-- Let's add a department without students and a student without a department
INSERT INTO Departments (DeptID, DeptName, HeadOfDept, Building) VALUES
('CH', 'Chemistry', 'Dr. Patel', 'Building C');
-- Student 105 (Frank) has DeptID 'PH' which doesn't exist in Departments
initially

-- JOIN Queries

-- 1. INNER JOIN: Students and their departments (only matching records)
SELECT
    s.FirstName,
    s.LastName,
    d.DeptName
FROM UniversityStudents s
INNER JOIN Departments d ON s.DeptID = d.DeptID;

-- 2. LEFT JOIN: All students and their departments (if any)
SELECT
    s.FirstName,
    s.LastName,
    d.DeptName
FROM UniversityStudents s
LEFT JOIN Departments d ON s.DeptID = d.DeptID;

-- 3. RIGHT JOIN: All departments and their students (if any)
-- Note: MySQL users might need to swap tables and use LEFT JOIN.
SELECT
```

```
    s.FirstName,
    s.LastName,
    d.DeptName
FROM UniversityStudents s
RIGHT JOIN Departments d ON s.DeptID = d.DeptID;

-- Equivalent for MySQL using LEFT JOIN:
-- SELECT
--     s.FirstName,
--     s.LastName,
--     d.DeptName
-- FROM Departments d
-- LEFT JOIN UniversityStudents s ON s.DeptID = d.DeptID;


-- 4. FULL OUTER JOIN: All students and all departments (matching or not)
-- Note: FULL OUTER JOIN is not directly supported in MySQL.
-- Emulate FULL OUTER JOIN using UNION of LEFT JOIN and RIGHT JOIN (or LEFT
JOIN and anti-join for right-only records)
SELECT
    s.FirstName,
    s.LastName,
    d.DeptName
FROM UniversityStudents s
LEFT JOIN Departments d ON s.DeptID = d.DeptID
UNION
SELECT
    s.FirstName,
    s.LastName,
    d.DeptName
FROM UniversityStudents s
RIGHT JOIN Departments d ON s.DeptID = d.DeptID
WHERE s.StudentID IS NULL; -- Only include rows from right join that didn't
match left join

-- For databases supporting FULL OUTER JOIN (e.g., PostgreSQL, Oracle, SQL
Server):
-- SELECT
--     s.FirstName,
--     s.LastName,
--     d.DeptName
-- FROM UniversityStudents s
-- FULL OUTER JOIN Departments d ON s.DeptID = d.DeptID;

-- 5. CROSS JOIN: Every student with every department (Cartesian product)
SELECT
    s.FirstName,
    d.DeptName
FROM UniversityStudents s
CROSS JOIN Departments d;

-- 6. SELF JOIN: Find students who are the same age
-- Assuming StudentID 101 (Alice, 22) and 102 (Bob, 22)
SELECT
    s1.FirstName AS Student1_FirstName,
    s1.LastName AS Student1_LastName,
    s2.FirstName AS Student2_FirstName,
    s2.LastName AS Student2_LastName,
    s1.Age
FROM UniversityStudents s1
INNER JOIN UniversityStudents s2 ON s1.Age = s2.Age AND s1.StudentID <
s2.StudentID; -- Avoid duplicate pairs
```

## Input

The data in `UniversityStudents` and `Departments` tables.

## Expected Output

- **INNER JOIN:** Only students who have a matching department, and departments that have matching students.

```
FirstName | LastName | DeptName
----------|----------|--------------------
Alice     | Smith    | Computer Science
Bob       | Johnson  | Electrical Engineering
Diana     | Prince   | Computer Science
```

- **LEFT JOIN:** All students, and their department names if a match exists. Frank Green will appear with a `NULL DeptName`.

```
FirstName | LastName | DeptName
----------|----------|--------------------
Alice     | Smith    | Computer Science
Bob       | Johnson  | Electrical Engineering
Diana     | Prince   | Computer Science
Frank     | Green    | NULL
```

- **RIGHT JOIN:** All departments, and their student names if a match exists. Chemistry (CH) and Mathematics (MA) will appear with `NULL` student names.

```
FirstName | LastName | DeptName
----------|----------|--------------------
Alice     | Smith    | Computer Science
Bob       | Johnson  | Electrical Engineering
Diana     | Prince   | Computer Science
NULL      | NULL     | Mathematics
NULL      | NULL     | Chemistry
```

- **FULL OUTER JOIN (emulated):** All students and all departments. Frank Green will have `NULL` for `DeptName`, and Chemistry and Mathematics will have `NULL` for student names.
- **CROSS JOIN:** Every student name paired with every department name. If there are 4 students and 4 departments, there will be 16 rows.
- **SELF JOIN:** Pairs of students who have the same age.

```
Student1_FirstName | Student1_LastName | Student2_FirstName | Student2_LastName | Age
-------------------|-------------------|--------------------|-------------------|----
Alice              | Smith             | Bob                | Johnson           | 22
```

(Assuming Alice and Bob are both 22)

# Lab 8: Correlated Subqueries

## Aim

To understand and implement correlated subqueries, where the inner query depends on the outer query for its execution.

## Procedure

1. **Connect to the Database:** Establish a connection to your SQL database.
2. **Identify Use Cases:** Understand when a correlated subquery is appropriate (e.g., finding records that have a certain property relative to other records in the same or another table).
3. **Implement Correlated Subqueries:** Write queries where the inner subquery references a column from the outer query. This typically involves `EXISTS`, `NOT EXISTS`, or aggregate functions in the subquery.

## Source Code

```
-- Assuming UniversityStudents and Departments tables from previous labs

-- 1. Find students whose age is greater than the average age of students in
their own major.
SELECT
    s.FirstName,
    s.LastName,
    s.Age,
    s.Major
FROM UniversityStudents s
WHERE s.Age > (SELECT AVG(s2.Age)
               FROM UniversityStudents s2
               WHERE s2.Major = s.Major);

-- 2. Find departments that have no students enrolled. (Using NOT EXISTS)
SELECT
    d.DeptName
FROM Departments d
WHERE NOT EXISTS (SELECT 1
                  FROM UniversityStudents s
                  WHERE s.DeptID = d.DeptID);

-- 3. Find students who are the oldest in their respective departments.
SELECT
    s.FirstName,
    s.LastName,
    s.Age,
    d.DeptName
FROM UniversityStudents s
JOIN Departments d ON s.DeptID = d.DeptID
WHERE s.Age = (SELECT MAX(s2.Age)
               FROM UniversityStudents s2
               WHERE s2.DeptID = s.DeptID);

-- 4. Find students who have the same last name as another student in a
different major.
SELECT
    s1.FirstName,
    s1.LastName,
    s1.Major
FROM UniversityStudents s1
WHERE EXISTS (SELECT 1
              FROM UniversityStudents s2
              WHERE s2.LastName = s1.LastName
```

```
                AND s2.Major <> s1.Major);
```

## Input

The data in `UniversityStudents` and `Departments` tables.

## Expected Output

- **Students older than average in their major:** A list of students whose age is above the average for their specific major.
    - Example: If CS average is 23, and Diana is 24, she would be listed.
- **Departments with no students:**
    - Example: If 'Chemistry' department has no students, it would be listed.
    - `DeptName`
    - `----------`
    - `Chemistry`

- **Oldest student in each department:** A list of students who have the maximum age within their department.
    - Example:
    - `FirstName | LastName | Age | DeptName`
    - `----------|----------|-----|--------------------`
    - `Diana     | Prince   | 24  | Computer Science`
    - `Bob       | Johnson  | 22  | Electrical Engineering`
    - `Charlie   | Brown    | 20  | Mathematics`
    - `Frank     | Green    | 21  | Physics`

- **Students with same last name but different major:** A list of students whose last name matches another student, but they are in different majors.
    - Example: If there were 'John Doe' in CS and 'Jane Doe' in EE.

# Lab 9: Decomposition using FD- dependency preservation

## Aim

To understand the concepts of functional dependencies (FDs), normalization, and decomposition of a relational schema into smaller, well-structured relations that are in higher normal forms, ensuring dependency preservation.

## Procedure

1. **Understand Functional Dependencies:** Identify FDs within a given relation schema. An FD A→B means that the value of A uniquely determines the value of B.
2. **Identify Anomalies:** Recognize insertion, deletion, and update anomalies in unnormalized relations.
3. **Normalization Process:** Apply normalization rules (e.g., to 3NF or BCNF) to decompose a relation.
4. **Dependency Preservation:** Ensure that all original functional dependencies can be inferred from the functional dependencies in the decomposed relations. This means that no FD is lost during the decomposition.
5. **Lossless Join Decomposition:** Ensure that joining the decomposed relations back together yields the original relation without spurious tuples.

## Source Code

This lab primarily involves theoretical understanding and schema design, rather than executable SQL code. However, we can demonstrate the DDL for the decomposed tables.

**Example Scenario:** Consider a relation `R(StudentID, StudentName, CourseID, CourseName, Instructor, InstructorOffice, Grade)` with the following functional dependencies:

- StudentID→StudentName**Error! Filename not specified.**
- CourseID→CourseName,Instructor**Error! Filename not specified.**
- Instructor→InstructorOffice**Error! Filename not specified.**
- (StudentID,CourseID)→Grade**Error! Filename not specified.**

This relation is not in 3NF because of transitive dependencies (CourseID→Instructor→InstructorOffice) and partial dependencies (CourseID→CourseName,Instructor where CourseID is part of the composite key (StudentID,CourseID)).

**Decomposition to 3NF (Dependency Preserving & Lossless Join):**

1. **Student (StudentID, StudentName)**
   - FD: StudentID→StudentName**Error! Filename not specified.**
2. **Course (CourseID, CourseName, Instructor)**
   - FD: CourseID→CourseName,Instructor**Error! Filename not specified.**
3. **Instructor (Instructor, InstructorOffice)**
   - FD: Instructor→InstructorOffice**Error! Filename not specified.**
4. **Enrollment (StudentID, CourseID, Grade)**
   - FD: (StudentID,CourseID)→Grade**Error! Filename not specified.**

## Input

A relational schema with a set of functional dependencies (conceptual input).

## Expected Output

The decomposed relational schemas (table structures) that are in 3NF (or BCNF) and preserve the original functional dependencies.

**SQL DDL for the Decomposed Tables:**

```
-- Original (unnormalized) conceptual table:
-- CREATE TABLE StudentCourseInfo (
--     StudentID INT,
--     StudentName VARCHAR(100),
--     CourseID VARCHAR(10),
--     CourseName VARCHAR(100),
--     Instructor VARCHAR(100),
--     InstructorOffice VARCHAR(50),
--     Grade VARCHAR(5),
--     PRIMARY KEY (StudentID, CourseID)
-- );

-- Decomposed Tables (3NF)

CREATE TABLE Student (
    StudentID INT PRIMARY KEY,
    StudentName VARCHAR(100) NOT NULL
);

CREATE TABLE Course (
    CourseID VARCHAR(10) PRIMARY KEY,
    CourseName VARCHAR(100) NOT NULL,
    Instructor VARCHAR(100) NOT NULL
);

CREATE TABLE Instructor (
    Instructor VARCHAR(100) PRIMARY KEY,
    InstructorOffice VARCHAR(50)
);

CREATE TABLE Enrollment (
    StudentID INT,
    CourseID VARCHAR(10),
    Grade VARCHAR(5),
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);

-- Note: In this decomposition, the Instructor column in the Course table
acts as a foreign key
-- to the Instructor table, even though it's not explicitly declared as such
in the DDL above
-- for simplicity. In a real system, you'd add:
-- ALTER TABLE Course ADD FOREIGN KEY (Instructor) REFERENCES
Instructor(Instructor);
-- (This would require Instructor to be a PRIMARY KEY in the Instructor
table, which it is here).
```

# Lab 10: PL/SQL Conditional and Iterative Statements

## Aim

To understand and implement conditional (IF-THEN-ELSE, CASE) and iterative (LOOP, WHILE, FOR) control structures in PL/SQL (Procedural Language/SQL) for writing more complex and dynamic database programs.

## Procedure

1. **Connect to Oracle/PL/SQL Environment:** Use a tool like SQL Developer, SQL*Plus, or a similar environment that supports PL/SQL.
2. **Declare Variables:** Declare variables to store data.
3. **Conditional Statements:**
    - `IF-THEN-END IF`: Execute a block of code if a condition is true.
    - `IF-THEN-ELSE-END IF`: Execute one block if true, another if false.
    - `IF-THEN-ELSIF-ELSE-END IF`: Handle multiple conditions.
    - `CASE` statement: Select one of many sequences of statements to execute based on a value.
4. **Iterative Statements (Loops):**
    - `LOOP-EXIT WHEN-END LOOP`: Basic loop that exits when a condition is met.
    - `WHILE-LOOP-END LOOP`: Loop that continues as long as a condition is true.
    - `FOR-LOOP-END LOOP`: Loop for a fixed number of iterations or for iterating over query results (cursor FOR loop).

## Source Code

```
-- Assuming an Oracle/PL/SQL environment

-- 1. Conditional Statements (IF-THEN-ELSIF-ELSE)
DECLARE
    p_score NUMBER := 85;
    v_grade VARCHAR2(10);
BEGIN
    IF p_score >= 90 THEN
        v_grade := 'A';
    ELSIF p_score >= 80 THEN
        v_grade := 'B';
    ELSIF p_score >= 70 THEN
        v_grade := 'C';
    ELSIF p_score >= 60 THEN
        v_grade := 'D';
    ELSE
        v_grade := 'F';
    END IF;
    DBMS_OUTPUT.PUT_LINE('Score: ' || p_score || ', Grade: ' || v_grade);
END;
/

-- 2. Conditional Statements (CASE)
DECLARE
    p_day_number NUMBER := 3;
    v_day_name VARCHAR2(20);
BEGIN
    CASE p_day_number
        WHEN 1 THEN v_day_name := 'Monday';
        WHEN 2 THEN v_day_name := 'Tuesday';
        WHEN 3 THEN v_day_name := 'Wednesday';
        WHEN 4 THEN v_day_name := 'Thursday';
        WHEN 5 THEN v_day_name := 'Friday';
```

```
        WHEN 6 THEN v_day_name := 'Saturday';
        WHEN 7 THEN v_day_name := 'Sunday';
        ELSE v_day_name := 'Invalid Day';
    END CASE;
    DBMS_OUTPUT.PUT_LINE('Day Number: ' || p_day_number || ', Day Name: ' ||
v_day_name);
END;
/

-- 3. Iterative Statements (Simple LOOP with EXIT WHEN)
DECLARE
    v_counter NUMBER := 1;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE('Loop Counter: ' || v_counter);
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 5;
    END LOOP;
END;
/

-- 4. Iterative Statements (WHILE LOOP)
DECLARE
    v_num NUMBER := 10;
BEGIN
    WHILE v_num > 0 LOOP
        DBMS_OUTPUT.PUT_LINE('While Loop: ' || v_num);
        v_num := v_num - 1;
    END LOOP;
END;
/

-- 5. Iterative Statements (FOR LOOP - Numeric Range)
BEGIN
    FOR i IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE('For Loop (Numeric): ' || i);
    END LOOP;
END;
/

-- 6. Iterative Statements (FOR LOOP - Cursor FOR Loop)
-- Assuming UniversityStudents table exists
BEGIN
    FOR student_rec IN (SELECT StudentID, FirstName, LastName, Major FROM
UniversityStudents WHERE Age > 20) LOOP
        DBMS_OUTPUT.PUT_LINE('Student ID: ' || student_rec.StudentID ||
                             ', Name: ' || student_rec.FirstName || ' ' ||
student_rec.LastName ||
                             ', Major: ' || student_rec.Major);
    END LOOP;
END;
/
```

## Input

No direct user input for these examples; variables are declared within the PL/SQL blocks. The Cursor FOR Loop uses data from the `UniversityStudents` table.

## Expected Output

The `DBMS_OUTPUT.PUT_LINE` statements will print messages to the console (or output window of your PL/SQL client) demonstrating the flow of control:

- **Conditional:** "Score: 85, Grade: B", "Day Number: 3, Day Name: Wednesday"
- **Loops:** Numbers 1 to 5 printed for simple and FOR loops, 10 down to 1 for WHILE loop. For the Cursor FOR Loop, it will list details of students older than 20 from your `UniversityStudents` table.

# Lab 11: PL/SQL Exceptional Handling

## Aim

To understand and implement exception handling mechanisms in PL/SQL to gracefully manage runtime errors and prevent program termination.

## Procedure

1. **Connect to Oracle/PL/SQL Environment:** Use a tool that supports PL/SQL.
2. **Understand Exceptions:** Learn about predefined exceptions (e.g., `NO_DATA_FOUND`, `TOO_MANY_ROWS`, `ZERO_DIVIDE`) and user-defined exceptions.
3. **Implement EXCEPTION Block:** Use the `EXCEPTION` block within a PL/SQL block to catch and handle errors.
4. **Raise Exceptions:** Use `RAISE` to explicitly raise exceptions.
5. **SQLCODE and SQLERRM:** Use `SQLCODE` to get the error number and `SQLERRM` to get the error message.

## Source Code

```
-- Assuming an Oracle/PL/SQL environment

-- 1. Handling Predefined Exception: NO_DATA_FOUND
DECLARE
    v_student_name VARCHAR2(100);
    p_student_id NUMBER := 999; -- A StudentID that does not exist
BEGIN
    SELECT FirstName || ' ' || LastName INTO v_student_name
    FROM UniversityStudents
    WHERE StudentID = p_student_id;

    DBMS_OUTPUT.PUT_LINE('Student Found: ' || v_student_name);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Error: No student found with ID ' ||
p_student_id);
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
END;
/

-- 2. Handling Predefined Exception: TOO_MANY_ROWS
DECLARE
    v_student_name VARCHAR2(100);
BEGIN
    -- This will cause TOO_MANY_ROWS if there are multiple students in
'Computer Science'
    SELECT FirstName || ' ' || LastName INTO v_student_name
    FROM UniversityStudents
    WHERE Major = 'Computer Science';

    DBMS_OUTPUT.PUT_LINE('Student Found: ' || v_student_name);

EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Error: Multiple students found for the given
major. Please refine query.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
END;
```

```
/

-- 3. Handling Predefined Exception: ZERO_DIVIDE
DECLARE
    v_numerator NUMBER := 10;
    v_denominator NUMBER := 0;
    v_result NUMBER;
BEGIN
    v_result := v_numerator / v_denominator;
    DBMS_OUTPUT.PUT_LINE('Result: ' || v_result);
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Error: Division by zero is not allowed.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
END;
/

-- 4. User-Defined Exception
DECLARE
    e_invalid_age EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_invalid_age, -20001); -- Associate with a custom
error number
    p_age NUMBER := 15;
BEGIN
    IF p_age < 18 THEN
        RAISE e_invalid_age; -- Raise the user-defined exception
    END IF;
    DBMS_OUTPUT.PUT_LINE('Age is valid: ' || p_age);
EXCEPTION
    WHEN e_invalid_age THEN
        DBMS_OUTPUT.PUT_LINE('Custom Error: Age ' || p_age || ' is too young
to enroll.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An unexpected error occurred (SQLCODE: ' ||
SQLCODE || ', SQLERRM: ' || SQLERRM || ')');
END;
/

-- 5. Propagating an exception (re-raising)
DECLARE
    v_student_name VARCHAR2(100);
    p_student_id NUMBER := 999;
BEGIN
    BEGIN -- Inner block
        SELECT FirstName || ' ' || LastName INTO v_student_name
        FROM UniversityStudents
        WHERE StudentID = p_student_id;
        DBMS_OUTPUT.PUT_LINE('Student Found: ' || v_student_name);
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Inner Block: No data found for ID ' ||
p_student_id);
            RAISE; -- Re-raise the current exception to the outer block
    END;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Outer Block: Caught the NO_DATA_FOUND exception
again.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Outer Block: An unexpected error occurred: ' ||
SQLERRM);
END;
/
```

## Input

No direct user input. The examples demonstrate different error conditions by using non-existent IDs, queries returning multiple rows, or invalid arithmetic operations.

## Expected Output

The `DBMS_OUTPUT.PUT_LINE` statements will show the error messages from the `EXCEPTION` blocks, indicating that the errors were caught and handled, preventing the program from crashing.

- **NO_DATA_FOUND:** "Error: No student found with ID 999"
- **TOO_MANY_ROWS:** "Error: Multiple students found for the given major. Please refine query."
- **ZERO_DIVIDE:** "Error: Division by zero is not allowed."
- **User-Defined Exception:** "Custom Error: Age 15 is too young to enroll."
- **Propagating Exception:** "Inner Block: No data found for ID 999", followed by "Outer Block: Caught the NO_DATA_FOUND exception again."

# Lab 12: PL/SQL Trigger

## Aim

To understand and implement database triggers in PL/SQL, which are stored programs that automatically execute (or "fire") in response to certain events on a table (e.g., `INSERT`, `UPDATE`, `DELETE`) or database-level events.

## Procedure

1. **Connect to Oracle/PL/SQL Environment:** Use a tool that supports PL/SQL.
2. **Understand Trigger Types:** Learn about different trigger types (row-level vs. statement-level, BEFORE vs. AFTER, DML vs. DDL vs. Database events).
3. **Create a Trigger:** Use the `CREATE TRIGGER` statement to define a trigger, specifying the timing, event, and the table it acts upon.
4. **Use OLD and NEW pseudo-records:** Access the old and new values of rows for row-level triggers (`:OLD.column_name`, `:NEW.column_name`).
5. **Test the Trigger:** Perform DML operations on the associated table to observe the trigger's behavior.

## Source Code

```
-- Assuming an Oracle/PL/SQL environment
-- Let's create an audit table to log changes to UniversityStudents
CREATE TABLE StudentAudit (
    AuditID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    StudentID NUMBER,
    OldMajor VARCHAR2(150),
    NewMajor VARCHAR2(150),
    ChangeDate TIMESTAMP,
    OperationType VARCHAR2(10)
);

-- 1. BEFORE INSERT Row-Level Trigger: Set default Major if not provided
CREATE OR REPLACE TRIGGER trg_before_insert_student
BEFORE INSERT ON UniversityStudents
FOR EACH ROW
```

```
BEGIN
    IF :NEW.Major IS NULL THEN
        :NEW.Major := 'Undeclared';
    END IF;
    -- Ensure StudentID is not null, if it's not auto-incrementing
    IF :NEW.StudentID IS NULL THEN
        SELECT NVL(MAX(StudentID), 0) + 1 INTO :NEW.StudentID FROM
UniversityStudents;
    END IF;
END;
/

-- 2. AFTER UPDATE Row-Level Trigger: Log changes to Major
CREATE OR REPLACE TRIGGER trg_after_update_major
AFTER UPDATE OF Major ON UniversityStudents
FOR EACH ROW
WHEN (OLD.Major IS DISTINCT FROM NEW.Major) -- Only fire if Major actually
changed
BEGIN
    INSERT INTO StudentAudit (StudentID, OldMajor, NewMajor, ChangeDate,
OperationType)
    VALUES (:OLD.StudentID, :OLD.Major, :NEW.Major, SYSTIMESTAMP, 'UPDATE');
END;
/

-- 3. AFTER DELETE Row-Level Trigger: Log deleted students
CREATE OR REPLACE TRIGGER trg_after_delete_student
AFTER DELETE ON UniversityStudents
FOR EACH ROW
BEGIN
    INSERT INTO StudentAudit (StudentID, OldMajor, NewMajor, ChangeDate,
OperationType)
    VALUES (:OLD.StudentID, :OLD.Major, NULL, SYSTIMESTAMP, 'DELETE');
END;
/

-- Test the Triggers

-- Test 1: BEFORE INSERT (insert a student with NULL major)
INSERT INTO UniversityStudents (FirstName, LastName, Age, Email, DeptID,
EnrollmentDate)
VALUES ('New', 'Student', 19, 'new.s@example.com', 'CS', SYSDATE);

-- Test 2: AFTER UPDATE (change a student's major)
UPDATE UniversityStudents
SET Major = 'Data Science'
WHERE StudentID = 101; -- Alice Smith

-- Test 3: AFTER DELETE (delete a student)
DELETE FROM UniversityStudents
WHERE StudentID = 102; -- Bob Johnson

-- Verify changes in StudentAudit table
SELECT * FROM StudentAudit;
```

## Input

DML operations (`INSERT`, `UPDATE`, `DELETE`) on the `UniversityStudents` table.

## Expected Output

- **BEFORE INSERT:** When a new student is inserted without a `Major`, the trigger will automatically set it to 'Undeclared'.

- **AFTER UPDATE:** An entry will be added to the `StudentAudit` table whenever a student's `Major` is changed, recording the old and new major, student ID, and timestamp.
- **AFTER DELETE:** An entry will be added to the `StudentAudit` table whenever a student record is deleted, logging the deleted student's ID and old major.
- The `SELECT * FROM StudentAudit;` query will show the audit trail created by the triggers.

# Lab 13: Authenticating the user (Users Credential ability)

## Aim

To understand and implement a basic user authentication mechanism within a database context, typically involving storing user credentials (username, hashed password) and validating them upon login attempts.

## Procedure

1. **Design User Table:** Create a table to store user information, including a username and a securely hashed password. **Never store plain text passwords.**
2. **Password Hashing:** Understand the importance of hashing passwords (e.g., using `DBMS_CRYPTO` in Oracle, `PASSWORD()` in MySQL, `crypt()` in PostgreSQL).
3. **User Registration (Conceptual):** Outline the process of inserting a new user with a hashed password.
4. **User Authentication Logic:** Develop a PL/SQL block or stored procedure that takes a username and a plain text password, hashes the provided password, and compares it with the stored hashed password.
5. **Return Authentication Status:** Provide a mechanism to indicate successful or failed authentication.

## Source Code

```
-- Assuming an Oracle/PL/SQL environment
-- For password hashing, Oracle's DBMS_CRYPTO package is used.
-- You might need to grant execute privileges on DBMS_CRYPTO to your user.
-- GRANT EXECUTE ON DBMS_CRYPTO TO your_user;

-- 1. Create User Credentials Table
CREATE TABLE AppUsers (
    UserID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    Username VARCHAR2(50) UNIQUE NOT NULL,
    PasswordHash VARCHAR2(256) NOT NULL, -- To store SHA-256 hash
    Salt VARCHAR2(64) -- Optional: for salting passwords
);

-- Function to generate a random salt (for better security)
CREATE OR REPLACE FUNCTION generate_salt RETURN VARCHAR2 IS
    v_salt_raw RAW(32); -- 32 bytes for a 64-character hex string
BEGIN
    v_salt_raw := DBMS_CRYPTO.RANDOMBYTES(32);
    RETURN RAWTOHEX(v_salt_raw);
END;
/

-- Function to hash a password with a salt
CREATE OR REPLACE FUNCTION hash_password (
    p_password IN VARCHAR2,
    p_salt     IN VARCHAR2
) RETURN VARCHAR2 IS
    v_hash RAW(256);
BEGIN
    v_hash := DBMS_CRYPTO.HASH(UTL_RAW.CAST_TO_RAW(p_password || p_salt),
DBMS_CRYPTO.HASH_SH256);
    RETURN RAWTOHEX(v_hash);
END;
/

-- Procedure for User Registration
```

```
CREATE OR REPLACE PROCEDURE register_user (
    p_username IN VARCHAR2,
    p_password IN VARCHAR2
) IS
    v_salt VARCHAR2(64);
    v_hashed_password VARCHAR2(256);
BEGIN
    v_salt := generate_salt;
    v_hashed_password := hash_password(p_password, v_salt);

    INSERT INTO AppUsers (Username, PasswordHash, Salt)
    VALUES (p_username, v_hashed_password, v_salt);

    DBMS_OUTPUT.PUT_LINE('User ' || p_username || ' registered
successfully.');
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE('Error: Username ' || p_username || ' already
exists.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error occurred during registration: ' ||
SQLERRM);
END;
/

-- Procedure for User Authentication
CREATE OR REPLACE PROCEDURE authenticate_user (
    p_username IN VARCHAR2,
    p_password IN VARCHAR2,
    p_authenticated OUT BOOLEAN
) IS
    v_stored_hash VARCHAR2(256);
    v_salt VARCHAR2(64);
    v_input_hash VARCHAR2(256);
BEGIN
    SELECT PasswordHash, Salt INTO v_stored_hash, v_salt
    FROM AppUsers
    WHERE Username = p_username;

    v_input_hash := hash_password(p_password, v_salt);

    IF v_input_hash = v_stored_hash THEN
        p_authenticated := TRUE;
        DBMS_OUTPUT.PUT_LINE('Authentication successful for user: ' ||
p_username);
    ELSE
        p_authenticated := FALSE;
        DBMS_OUTPUT.PUT_LINE('Authentication failed for user: ' || p_username
|| ' (Incorrect password).');
    END IF;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        p_authenticated := FALSE;
        DBMS_OUTPUT.PUT_LINE('Authentication failed: User ' || p_username ||
' not found.');
    WHEN OTHERS THEN
        p_authenticated := FALSE;
        DBMS_OUTPUT.PUT_LINE('An error occurred during authentication: ' ||
SQLERRM);
END;
/

-- Test the authentication system

-- 1. Register a new user
```

```
BEGIN
    register_user('testuser', 'mysecurepassword123');
    register_user('admin', 'adminpass');
END;
/

-- 2. Attempt to authenticate
DECLARE
    is_auth BOOLEAN;
BEGIN
    authenticate_user('testuser', 'mysecurepassword123', is_auth);
    authenticate_user('testuser', 'wrongpassword', is_auth);
    authenticate_user('nonexistentuser', 'anypass', is_auth);
    authenticate_user('admin', 'adminpass', is_auth);
END;
/
```

## Input

- For `register_user`: A username and a plain text password.
- For `authenticate_user`: A username and a plain text password.

## Expected Output

The `DBMS_OUTPUT.PUT_LINE` statements will indicate the success or failure of user registration and authentication attempts.

- "User testuser registered successfully."
- "Authentication successful for user: testuser"
- "Authentication failed for user: testuser (Incorrect password)."
- "Authentication failed: User nonexistentuser not found."

# Lab 14: Implementation by Using Tools Frontend (VB 10.0) and Backend (Oracle12g)

## Aim

To gain practical experience in building a simple database application by integrating a frontend (Visual Basic 10.0) with a backend database (Oracle 12c). This involves connecting to the database, performing DML operations, and displaying data in a user interface.

## Procedure

1. **Set up Oracle 12c Database:** Ensure Oracle 12c is installed and running, and you have a user with necessary table privileges (e.g., `UniversityStudents` table).
2. **Install Visual Studio 2010 (or later with VB.NET):** Install the IDE for Visual Basic development.
3. **Create a New VB.NET Project:** Start a new Windows Forms Application project in Visual Studio.
4. **Add Oracle Data Provider for .NET (ODP.NET):** This is crucial for connecting VB.NET to Oracle. Install it if not already present (often part of Oracle Client installation).
5. **Design User Interface:** Create a simple form with controls like TextBoxes for input, Buttons for actions (Insert, Update, Delete, Select), and a DataGridView to display results.
6. **Write VB.NET Code:**
    - **Database Connection:** Establish a connection string to your Oracle database.
    - **SQL Commands:** Write ADO.NET code to execute SQL `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements.
    - **Data Binding:** Populate the DataGridView with data retrieved from the database.
    - **Error Handling:** Implement `Try-Catch` blocks for database operations.

## Source Code (Conceptual - VB.NET)

This section provides conceptual VB.NET code snippets as a full, runnable VB.NET project is outside the scope of this text-based response.

**1. Project Setup:**

- Add a reference to `Oracle.DataAccess.dll` (or `Oracle.ManagedDataAccess.dll` for managed ODP.NET).

**2. Form Design (Example Controls):**

- `txtStudentID` (TextBox)
- `txtFirstName` (TextBox)
- `txtLastName` (TextBox)
- `txtMajor` (TextBox)
- `btnInsert` (Button)
- `btnUpdate` (Button)
- `btnDelete` (Button)
- `btnRefresh` (Button)
- `dgvStudents` (DataGridView)

### 3. VB.NET Code (Form1.vb)

```
' Imports for Oracle Data Access
Imports Oracle.DataAccess.Client ' Or Oracle.ManagedDataAccess.Client

Public Class Form1

    ' Connection String (Replace with your Oracle details)
    Private Const connectionString As String = "Data
Source=(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=localhost)(PORT=1521))(CONNE
CT_DATA=(SERVICE_NAME=ORCL)));User Id=your_username;Password=your_password;"

    Private Sub Form1_Load(sender As Object, e As EventArgs) Handles
MyBase.Load
        LoadStudents()
    End Sub

    Private Sub LoadStudents()
        Using conn As New OracleConnection(connectionString)
            Try
                conn.Open()
                Dim cmd As New OracleCommand("SELECT StudentID, FirstName,
LastName, Major FROM UniversityStudents ORDER BY StudentID", conn)
                Dim adapter As New OracleDataAdapter(cmd)
                Dim dt As New DataTable()
                adapter.Fill(dt)
                dgvStudents.DataSource = dt
            Catch ex As OracleException
                MessageBox.Show("Error loading students: " & ex.Message,
"Database Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
            Catch ex As Exception
                MessageBox.Show("An unexpected error occurred: " &
ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
            End Try
        End Using
    End Sub

    Private Sub btnInsert_Click(sender As Object, e As EventArgs) Handles
btnInsert.Click
        Using conn As New OracleConnection(connectionString)
            Try
                conn.Open()
                Dim sql As String = "INSERT INTO UniversityStudents
(StudentID, FirstName, LastName, Major) VALUES (:p_sid, :p_fname, :p_lname,
:p_major)"
                Dim cmd As New OracleCommand(sql, conn)

                cmd.Parameters.Add("p_sid", OracleDbType.Int32).Value =
Integer.Parse(txtStudentID.Text)
                cmd.Parameters.Add("p_fname", OracleDbType.Varchar2).Value =
txtFirstName.Text
                cmd.Parameters.Add("p_lname", OracleDbType.Varchar2).Value =
txtLastName.Text
                cmd.Parameters.Add("p_major", OracleDbType.Varchar2).Value =
txtMajor.Text

                cmd.ExecuteNonQuery()
                MessageBox.Show("Student inserted successfully!", "Success",
MessageBoxButtons.OK, MessageBoxIcon.Information)
                ClearForm()
                LoadStudents()
            Catch ex As OracleException
                MessageBox.Show("Error inserting student: " & ex.Message,
"Database Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
            Catch ex As Exception
```

```
                MessageBox.Show("An unexpected error occurred: " &
ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
            End Try
        End Using
    End Sub

    Private Sub btnUpdate_Click(sender As Object, e As EventArgs) Handles
btnUpdate.Click
        Using conn As New OracleConnection(connectionString)
            Try
                conn.Open()
                Dim sql As String = "UPDATE UniversityStudents SET FirstName
= :p_fname, LastName = :p_lname, Major = :p_major WHERE StudentID = :p_sid"
                Dim cmd As New OracleCommand(sql, conn)

                cmd.Parameters.Add("p_fname", OracleDbType.Varchar2).Value =
txtFirstName.Text
                cmd.Parameters.Add("p_lname", OracleDbType.Varchar2).Value =
txtLastName.Text
                cmd.Parameters.Add("p_major", OracleDbType.Varchar2).Value =
txtMajor.Text
                cmd.Parameters.Add("p_sid", OracleDbType.Int32).Value =
Integer.Parse(txtStudentID.Text)

                Dim rowsAffected As Integer = cmd.ExecuteNonQuery()
                If rowsAffected > 0 Then
                    MessageBox.Show("Student updated successfully!",
"Success", MessageBoxButtons.OK, MessageBoxIcon.Information)
                Else
                    MessageBox.Show("No student found with that ID to
update.", "Info", MessageBoxButtons.OK, MessageBoxIcon.Information)
                End If
                ClearForm()
                LoadStudents()
            Catch ex As OracleException
                MessageBox.Show("Error updating student: " & ex.Message,
"Database Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
            Catch ex As Exception
                MessageBox.Show("An unexpected error occurred: " &
ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
            End Try
        End Using
    End Sub

    Private Sub btnDelete_Click(sender As Object, e As EventArgs) Handles
btnDelete.Click
        If String.IsNullOrWhiteSpace(txtStudentID.Text) Then
            MessageBox.Show("Please enter Student ID to delete.", "Warning",
MessageBoxButtons.OK, MessageBoxIcon.Warning)
            Return
        End If

        If MessageBox.Show("Are you sure you want to delete this student?",
"Confirm Delete", MessageBoxButtons.YesNo, MessageBoxIcon.Question) =
DialogResult.Yes Then
            Using conn As New OracleConnection(connectionString)
                Try
                    conn.Open()
                    Dim sql As String = "DELETE FROM UniversityStudents WHERE
StudentID = :p_sid"
                    Dim cmd As New OracleCommand(sql, conn)
                    cmd.Parameters.Add("p_sid", OracleDbType.Int32).Value =
Integer.Parse(txtStudentID.Text)

                    Dim rowsAffected As Integer = cmd.ExecuteNonQuery()
                    If rowsAffected > 0 Then
```

```
                        MessageBox.Show("Student deleted successfully!",
"Success", MessageBoxButtons.OK, MessageBoxIcon.Information)
                    Else
                        MessageBox.Show("No student found with that ID to
delete.", "Info", MessageBoxButtons.OK, MessageBoxIcon.Information)
                    End If
                    ClearForm()
                    LoadStudents()
                Catch ex As OracleException
                    MessageBox.Show("Error deleting student: " & ex.Message,
"Database Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
                Catch ex As Exception
                    MessageBox.Show("An unexpected error occurred: " &
ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error)
                End Try
            End Using
        End If
    End Sub

    Private Sub btnRefresh_Click(sender As Object, e As EventArgs) Handles
btnRefresh.Click
        LoadStudents()
    End Sub

    Private Sub ClearForm()
        txtStudentID.Clear()
        txtFirstName.Clear()
        txtLastName.Clear()
        txtMajor.Clear()
    End Sub

    Private Sub dgvStudents_CellClick(sender As Object, e As
DataGridViewCellEventArgs) Handles dgvStudents.CellClick
        If e.RowIndex >= 0 Then
            Dim row As DataGridViewRow = dgvStudents.Rows(e.RowIndex)
            txtStudentID.Text = row.Cells("StudentID").Value.ToString()
            txtFirstName.Text = row.Cells("FirstName").Value.ToString()
            txtLastName.Text = row.Cells("LastName").Value.ToString()
            txtMajor.Text = row.Cells("Major").Value.ToString()
        End If
    End Sub

End Class
```

## Input

User input through the VB.NET application's text boxes and button clicks.

## Expected Output

- A functional Windows Forms application that connects to an Oracle 12c database.
- The `DataGridView` will display records from the `UniversityStudents` table.
- Users can enter data into text boxes and click "Insert" to add new student records.
- Users can select a row in the `DataGridView`, modify details in text boxes, and click "Update" to modify existing records.
- Users can enter a Student ID and click "Delete" to remove a record.
- Appropriate success or error messages will be displayed in message boxes.

# Lab 15 Project:

## i) Employee payroll processing system

## ii) Student Marksheet processing system

## iii) Banking system

## Aim

To apply the knowledge gained from previous labs to develop a comprehensive database-driven application for a real-world scenario. This project will involve designing a database schema, implementing DDL and DML operations, utilizing PL/SQL for business logic, and potentially integrating a frontend for user interaction.

## Procedure

For each project, the general procedure will be:

1. **Requirement Analysis:** Understand the core functionalities, entities, and relationships for the chosen system (e.g., for Employee Payroll: Employees, Departments, Salary, Attendance, Deductions).
2. **ER Diagram Design:** Create a detailed Entity-Relationship Diagram for the system.
3. **Relational Schema Design:** Translate the ER Diagram into a set of normalized relational tables with primary and foreign keys.
4. **Database Implementation (DDL):** Write SQL `CREATE TABLE` statements for all tables, including constraints.
5. **Data Population (DML):** Insert sample data into the tables.
6. **PL/SQL Business Logic:**
    - Develop stored procedures for common operations (e.g., calculating payroll, generating marksheet, processing transactions).
    - Implement functions for calculations.
    - Create triggers for auditing or enforcing business rules.
    - Incorporate exception handling for robust operations.
7. **Query Development:** Write complex SQL queries for reporting and data retrieval.
8. **Frontend Development (Optional but Recommended):** If time and tools permit, create a simple user interface (e.g., using VB.NET, Python with Tkinter/PyQt, or a web framework) to interact with the database.

## Source Code (Conceptual - SQL & PL/SQL)

Due to the scope of a full project, only conceptual SQL and PL/SQL structures are provided.

### *i) Employee Payroll Processing System*

**Database Schema (Example Tables):**

```
CREATE TABLE Departments (
    DeptID VARCHAR(10) PRIMARY KEY,
    DeptName VARCHAR(100) NOT NULL UNIQUE
);

CREATE TABLE Employees (
    EmployeeID NUMBER PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
```

```
    LastName VARCHAR(50) NOT NULL,
    DeptID VARCHAR(10),
    DateOfJoining DATE,
    BasicSalary DECIMAL(10, 2),
    FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)
);

CREATE TABLE Payroll (
    PayrollID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    EmployeeID NUMBER,
    PayPeriodStart DATE,
    PayPeriodEnd DATE,
    GrossSalary DECIMAL(10, 2),
    Deductions DECIMAL(10, 2),
    NetSalary DECIMAL(10, 2),
    PayrollDate DATE DEFAULT SYSDATE,
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
);

-- Other tables like Attendance, Allowances, Deductions, etc.
```

**PL/SQL Procedure (Example: Calculate Monthly Payroll)**

```
CREATE OR REPLACE PROCEDURE calculate_monthly_payroll (
    p_month IN NUMBER,
    p_year  IN NUMBER
) IS
    CURSOR emp_cursor IS
        SELECT EmployeeID, BasicSalary
        FROM Employees;
    v_gross_salary DECIMAL(10, 2);
    v_deductions DECIMAL(10, 2) := 0; -- Simplified, would involve more logic
    v_net_salary DECIMAL(10, 2);
    v_pay_period_start DATE;
    v_pay_period_end DATE;
BEGIN
    v_pay_period_start := TRUNC(ADD_MONTHS(SYSDATE, -(TO_CHAR(SYSDATE, 'MM')
- p_month)), 'MONTH');
    v_pay_period_end := LAST_DAY(v_pay_period_start);

    FOR emp_rec IN emp_cursor LOOP
        -- Calculate gross salary (simplified: BasicSalary + some allowance)
        v_gross_salary := emp_rec.BasicSalary + (emp_rec.BasicSalary * 0.10);
-- 10% allowance

        -- Calculate deductions (simplified: 5% tax)
        v_deductions := v_gross_salary * 0.05;

        v_net_salary := v_gross_salary - v_deductions;

        INSERT INTO Payroll (EmployeeID, PayPeriodStart, PayPeriodEnd,
GrossSalary, Deductions, NetSalary)
        VALUES (emp_rec.EmployeeID, v_pay_period_start, v_pay_period_end,
v_gross_salary, v_deductions, v_net_salary);

        DBMS_OUTPUT.PUT_LINE('Processed payroll for Employee ' ||
emp_rec.EmployeeID || '. Net Salary: ' || v_net_salary);
    END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error calculating payroll: ' || SQLERRM);
END;
/
```

*ii) Student Marksheet Processing System*

**Database Schema (Example Tables):**

```
CREATE TABLE Students (
    StudentID NUMBER PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DateOfBirth DATE,
    Major VARCHAR(100)
);

CREATE TABLE Courses (
    CourseID VARCHAR(10) PRIMARY KEY,
    CourseName VARCHAR(100) NOT NULL,
    Credits NUMBER
);

CREATE TABLE Enrollments (
    EnrollmentID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    StudentID NUMBER,
    CourseID VARCHAR(10),
    EnrollmentDate DATE,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);

CREATE TABLE Grades (
    GradeID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    EnrollmentID NUMBER,
    Score DECIMAL(5, 2),
    Letter_Grade VARCHAR(2),
    FOREIGN KEY (EnrollmentID) REFERENCES Enrollments(EnrollmentID)
);
```

**PL/SQL Procedure (Example: Generate Marksheet for a Student)**

```
CREATE OR REPLACE PROCEDURE generate_marksheet (
    p_student_id IN NUMBER
) IS
    v_student_name VARCHAR2(100);
    v_total_credits NUMBER := 0;
    v_total_score_points NUMBER := 0;
    v_gpa DECIMAL(4, 2);
BEGIN
    SELECT FirstName || ' ' || LastName INTO v_student_name
    FROM Students
    WHERE StudentID = p_student_id;

    DBMS_OUTPUT.PUT_LINE('--- Marksheet for ' || v_student_name || ' (ID: '
|| p_student_id || ') ---');
    DBMS_OUTPUT.PUT_LINE('Course Name | Credits | Score | Grade');
    DBMS_OUTPUT.PUT_LINE('-----------------------------------------');

    FOR rec IN (SELECT c.CourseName, c.Credits, g.Score, g.Letter_Grade
                FROM Enrollments e
                JOIN Courses c ON e.CourseID = c.CourseID
                JOIN Grades g ON e.EnrollmentID = g.EnrollmentID
                WHERE e.StudentID = p_student_id) LOOP
        DBMS_OUTPUT.PUT_LINE(RPAD(rec.CourseName, 15) || ' | ' ||
                             RPAD(rec.Credits, 7) || ' | ' ||
                             RPAD(rec.Score, 5) || ' | ' ||
                             rec.Letter_Grade);
```

```
        v_total_credits := v_total_credits + rec.Credits;
        -- Simplified GPA calculation (e.g., A=4, B=3, C=2, D=1, F=0)
        CASE rec.Letter_Grade
            WHEN 'A' THEN v_total_score_points := v_total_score_points +
(rec.Credits * 4);
            WHEN 'B' THEN v_total_score_points := v_total_score_points +
(rec.Credits * 3);
            WHEN 'C' THEN v_total_score_points := v_total_score_points +
(rec.Credits * 2);
            WHEN 'D' THEN v_total_score_points := v_total_score_points +
(rec.Credits * 1);
            ELSE NULL;
        END CASE;
    END LOOP;

    IF v_total_credits > 0 THEN
        v_gpa := v_total_score_points / v_total_credits;
        DBMS_OUTPUT.PUT_LINE('-----------------------------------------
');
        DBMS_OUTPUT.PUT_LINE('Total Credits: ' || v_total_credits || ', GPA:
' || ROUND(v_gpa, 2));
    ELSE
        DBMS_OUTPUT.PUT_LINE('No courses found for this student.');
    END IF;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Student with ID ' || p_student_id || ' not
found.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error generating marksheet: ' || SQLERRM);
END;
/
```

*iii) Banking System*

**Database Schema (Example Tables):**

```
CREATE TABLE Customers (
    CustomerID NUMBER PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    DateOfBirth DATE,
    Address VARCHAR(255)
);

CREATE TABLE Accounts (
    AccountNumber VARCHAR(20) PRIMARY KEY,
    CustomerID NUMBER,
    AccountType VARCHAR(20) CHECK (AccountType IN ('Savings', 'Checking',
'Loan')),
    Balance DECIMAL(15, 2) NOT NULL,
    OpeningDate DATE DEFAULT SYSDATE,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE Transactions (
    TransactionID NUMBER GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    AccountNumber VARCHAR(20),
    TransactionDate TIMESTAMP DEFAULT SYSTIMESTAMP,
    TransactionType VARCHAR(20) CHECK (TransactionType IN ('Deposit',
'Withdrawal', 'Transfer')),
    Amount DECIMAL(15, 2) NOT NULL,
    Description VARCHAR(255),
    FOREIGN KEY (AccountNumber) REFERENCES Accounts(AccountNumber)
```

```
);
```

### PL/SQL Procedure (Example: Process Transaction - Deposit/Withdrawal)

```
CREATE OR REPLACE PROCEDURE process_transaction (
    p_account_number IN VARCHAR2,
    p_amount         IN DECIMAL,
    p_transaction_type IN VARCHAR2, -- 'Deposit' or 'Withdrawal'
    p_description    IN VARCHAR2 DEFAULT NULL
) IS
    v_current_balance DECIMAL(15, 2);
    e_insufficient_funds EXCEPTION;
BEGIN
    -- Lock the row to prevent concurrent updates
    SELECT Balance INTO v_current_balance
    FROM Accounts
    WHERE AccountNumber = p_account_number
    FOR UPDATE OF Balance NOWAIT; -- NOWAIT: return error if locked

    IF p_transaction_type = 'Deposit' THEN
        v_current_balance := v_current_balance + p_amount;
    ELSIF p_transaction_type = 'Withdrawal' THEN
        IF v_current_balance < p_amount THEN
            RAISE e_insufficient_funds;
        END IF;
        v_current_balance := v_current_balance - p_amount;
    ELSE
        RAISE_APPLICATION_ERROR(-20001, 'Invalid transaction type: ' ||
p_transaction_type);
    END IF;

    UPDATE Accounts
    SET Balance = v_current_balance
    WHERE AccountNumber = p_account_number;

    INSERT INTO Transactions (AccountNumber, TransactionType, Amount,
Description)
    VALUES (p_account_number, p_transaction_type, p_amount, p_description);

    COMMIT; -- Commit the transaction

    DBMS_OUTPUT.PUT_LINE(p_transaction_type || ' of ' || p_amount || '
processed for account ' || p_account_number || '. New balance: ' ||
v_current_balance);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error: Account ' || p_account_number || ' not
found.');
    WHEN e_insufficient_funds THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Error: Insufficient funds in account ' ||
p_account_number || ' for withdrawal of ' || p_amount);
    WHEN OTHERS THEN
        ROLLBACK; -- Rollback on any other error
        DBMS_OUTPUT.PUT_LINE('An error occurred during transaction: ' ||
SQLERRM);
END;
/
```

## Input

- **Employee Payroll:** Employee data, pay period details.
- **Student Marksheet:** Student data, course data, enrollment data, grades.
- **Banking System:** Customer data, account data, transaction details (account number, amount, type).

## Expected Output

- **Employee Payroll:** Calculated payroll records inserted into the `Payroll` table, and reports showing employee salaries.
- **Student Marksheet:** Generated marksheets for individual students, potentially with GPA calculations.
- **Banking System:** Updated account balances, recorded transactions, and appropriate messages for deposits, withdrawals, or errors (e.g., insufficient funds).