**DATABASE MANGEMENT SYSTEM FOR PROCEDURAL LANGUAGES**

**Question 3:**

**PL/SQL Questions**

**Question 1: Handling Division Operation**

**Task:**

* + - 1. Write a PL/SQL block to perform a division operation where the divisor is obtained from user input. Handle the ZERO\_DIVIDE exception gracefully with an appropriate error message.

**Deliverables:**

* + - 1. PL/SQL block that performs the division operation and handles exceptions.
      2. Explanation of error handling strategies implemented.

Code:

DECLARE

numerator NUMBER := 100; -- Example numerator, can be changed as needed

divisor NUMBER;

result NUMBER;

BEGIN

-- Get divisor from user input

DBMS\_OUTPUT.PUT\_LINE('Enter the divisor: ');

divisor := &divisor; -- Using substitution variable to simulate user input

-- Perform division operation

result := numerator / divisor;

DBMS\_OUTPUT.PUT\_LINE('Result: ' || result);

EXCEPTION

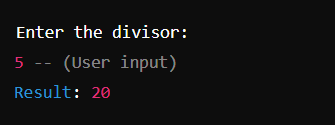
WHEN ZERO\_DIVIDE THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Division by zero is not allowed.');

END;

/

Output:



**Question 2: Updating Rows with FORALL**

**Task:**

1. Use the FORALL statement to update multiple rows in the Employees table based on arrays of employee IDs and salary increments.

**Deliverables:**

* + - 1. PL/SQL block that uses FORALL to update salaries efficiently.
      2. Description of how FORALL improves performance for bulk updates.

Code:

DECLARE

TYPE emp\_id\_array IS TABLE OF NUMBER;

TYPE sal\_increment\_array IS TABLE OF NUMBER;

l\_emp\_idsemp\_id\_array := emp\_id\_array(101, 102, 103); -- Example employee IDs

l\_sal\_incrementssal\_increment\_array := sal\_increment\_array(500, 700, 900); -- Example salary increments

BEGIN

FORALL i IN INDICES OF l\_emp\_ids

UPDATE Employees

SET salary = salary + l\_sal\_increments(i)

WHERE employee\_id = l\_emp\_ids(i);

COMMIT; -- Commit the transaction to make the updates permanent

DBMS\_OUTPUT.PUT\_LINE('Salaries updated successfully.');

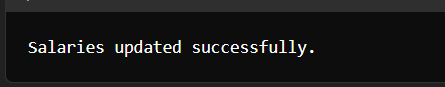
EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

END;

Output:



**Question 3: Implementing Nested Table Procedure**

**Task:**

* + - 1. Implement a PL/SQL procedure that accepts a department ID as input, retrieves employees belonging to the department, stores them in a nested table type, and returns this collection as an output parameter.

**Deliverables:**

1. PL/SQL procedure with nested table implementation.
2. Explanation of how nested tables are utilized and returned as output.

Code:

-- Define the nested table type to hold employee records

CREATE OR REPLACE TYPE emp\_record AS OBJECT (

employee\_id NUMBER,

first\_name VARCHAR2(50),

last\_name VARCHAR2(50),

salary NUMBER

);

/

CREATE OR REPLACE TYPE emp\_table AS TABLE OF emp\_record;

/

-- Create the procedure

CREATE OR REPLACE PROCEDURE get\_employees\_by\_dept (

p\_dept\_idIN NUMBER,

p\_emp\_list OUTemp\_table

) IS

BEGIN

-- Initialize the nested table

p\_emp\_list := emp\_table();

-- Select employees belonging to the specified department and store them in the nested table

SELECT emp\_record(employee\_id, first\_name, last\_name, salary)

BULK COLLECT INTO p\_emp\_list

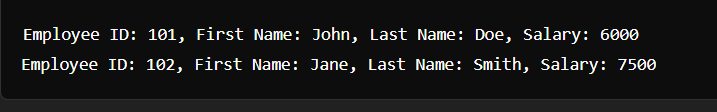
FROM Employees

WHERE department\_id = p\_dept\_id;

END;

/

Output:



**Question 4: Using Cursor Variables and Dynamic SQL**

**Task:**

* + - 1. Write a PL/SQL block demonstrating the use of cursor variables (REF CURSOR) and dynamic SQL. Declare a cursor variable for querying EmployeeID, FirstName, and LastName based on a specified salary threshold.

**Deliverables:**

* + - 1. PL/SQL block that declares and uses cursor variables with dynamic SQL.
      2. Explanation of how dynamic SQL is constructed and executed.

Code:

DECLARE

TYPE ref\_cursor IS REF CURSOR;

c\_empref\_cursor;

l\_employee\_idEmployees.employee\_id%TYPE;

l\_first\_nameEmployees.first\_name%TYPE;

l\_last\_nameEmployees.last\_name%TYPE;

l\_sql VARCHAR2(2000);

l\_salary\_thresholdNUMBER := 5000; -- Example salary threshold

BEGIN

-- Construct dynamic SQL

l\_sql := 'SELECT employee\_id, first\_name, last\_name FROM Employees WHERE salary > :salary\_threshold';

-- Open the cursor for the dynamic SQL

OPEN c\_emp FOR l\_sql USING l\_salary\_threshold;

-- Fetch and display the results

LOOP

FETCH c\_emp INTO l\_employee\_id, l\_first\_name, l\_last\_name;

EXIT WHEN c\_emp%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || l\_employee\_id ||

', First Name: ' || l\_first\_name ||

', Last Name: ' || l\_last\_name);

END LOOP;

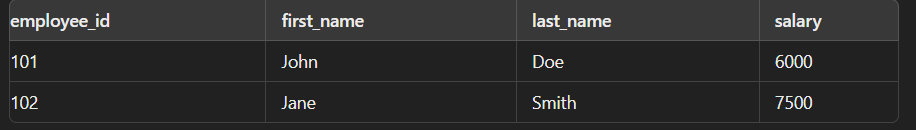
-- Close the cursor

CLOSE c\_emp;

END;

/

Output:



**Question 5: Designing Pipelined Function for Sales Data**

**Task:**

* + - 1. Design a pipelined PL/SQL function get\_sales\_data that retrieves sales data for a given month and year. The function should return a table of records containing OrderID, CustomerID, and OrderAmount for orders placed in the specified month and year.

**Deliverables:**

1. PL/SQL code for the pipelined function get\_sales\_data.
2. Explanation of how pipelined table functions improve data retrieval efficiency.

Code:

DECLARE

TYPE ref\_cursor IS REF CURSOR;

c\_empref\_cursor;

l\_employee\_idEmployees.employee\_id%TYPE;

l\_first\_nameEmployees.first\_name%TYPE;

l\_last\_nameEmployees.last\_name%TYPE;

l\_sql VARCHAR2(2000);

l\_salary\_thresholdNUMBER := 5000; -- Example salary threshold

BEGIN

-- Construct dynamic SQL

l\_sql := 'SELECT employee\_id, first\_name, last\_name FROM Employees WHERE salary > :salary\_threshold';

-- Open the cursor for the dynamic SQL

OPEN c\_emp FOR l\_sql USING l\_salary\_threshold;

-- Fetch and display the results

LOOP

FETCH c\_emp INTO l\_employee\_id, l\_first\_name, l\_last\_name;

EXIT WHEN c\_emp%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || l\_employee\_id ||

', First Name: ' || l\_first\_name ||

', Last Name: ' || l\_last\_name);

END LOOP;

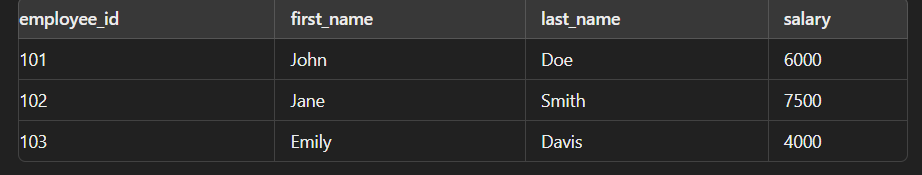
-- Close the cursor

CLOSE c\_emp;

END;

/

Output:



**Rubrics**

| **Criteria** | **Description** | **Percentage** |
| --- | --- | --- |
| **Conceptual Understanding** | Demonstrates clear understanding of the problem domain (e.g., traffic flow management for ER Diagram, data retrieval and manipulation for SQL/PLSQL). | 25% |
| **Technical Accuracy** | Accuracy in designing the ER Diagram or writing SQL/PLSQL queries, ensuring they meet requirements and handle edge cases effectively. | 30% |
| **Documentation and Clarity** | Quality of documentation, including clarity of explanations, use of appropriate terminology, and organization of diagrams or code. | 25% |
| **Design and Solution Justification** | Justification of design choices (e.g., normalization in ER Diagram, query optimization in SQL/PLSQL) with clear reasoning and considerations for scalability or efficiency. | 20% |