Engineer to Excel

SAVEETHA
SCHOOL OF ENGINEERING
Affliated to AICTE | IET-UK Accreditation

Assignment - 3

SAVEETHA SCHOOL OF ENGINEERING

Submitted by

VIMAL MAHENDRAN (192311068)

Submitted to

Dr. Christy Melwyn
Professor

Course Code: **CSA0556**

Course Name: **Database Management Systems for Relational Database**

Question 1: Handling Division Operation

Task:
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.

PL/SQL Block:

```
DECLARE
  numerator NUMBER := 100; -- You can replace this with any value or input
  divisor NUMBER;
  result NUMBER;
BEGIN
  -- Obtain the divisor from user input or another source
  divisor := :divisor_input; -- Replace :divisor_input with actual input mechanism

  -- Perform the 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.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An unexpected error occurred: ' || SQLERRM);
END;
```

Explanation of Error Handling Strategies:
- The `ZERO_DIVIDE` exception is specifically caught to handle cases where the divisor is zero, preventing the program from crashing and providing a user-friendly message.

- The `WHEN OTHERS` exception handler is a catch-all for any unexpected errors, ensuring that the program gracefully handles any unforeseen issues.

Question 2: Updating Rows with FORALL

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

PL/SQL Block:

```
DECLARE
  TYPE NumArray IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  emp_ids NumArray := NumArray(1001, 1002, 1003); -- Example employee IDs
  increments NumArray := NumArray(500, 600, 700); -- Corresponding salary increments
BEGIN
  FORALL i IN emp_ids.FIRST..emp_ids.LAST
    UPDATE Employees
    SET Salary = Salary + increments(i)
    WHERE EmployeeID = emp_ids(i);

  DBMS_OUTPUT.PUT_LINE('Salaries updated successfully.');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
```

Description of How FORALL Improves Performance:

- The `FORALL` statement allows for bulk binding, which reduces context switching between the PL/SQL and SQL engines. This leads to significant performance improvements when updating or inserting large numbers of rows.

Question 3: Implementing Nested Table Procedure

Task:
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.

PL/SQL Block:

```
CREATE OR REPLACE TYPE EmpRecord IS OBJECT (
  EmployeeID NUMBER,
  FirstName VARCHAR2(50),
  LastName VARCHAR2(50)
);

CREATE OR REPLACE TYPE EmpTable IS TABLE OF EmpRecord;

CREATE OR REPLACE PROCEDURE GetEmployeesByDept(
  dept_id IN NUMBER,
  emp_list OUT EmpTable
) IS
BEGIN
  SELECT EmpRecord(EmployeeID, FirstName, LastName)
  BULK COLLECT INTO emp_list
  FROM Employees
```

```
  WHERE DepartmentID = dept_id;

  IF emp_list.COUNT = 0 THEN
    DBMS_OUTPUT.PUT_LINE('No employees found for the given department ID.');
  END IF;
END;
```

Explanation of How Nested Tables Are Utilized:

- Nested tables are used to store collections of employee records, allowing for complex data structures within PL/SQL. This procedure retrieves and stores employee data in a nested table type and returns it as an output parameter, enabling the caller to access the data in a structured format.

Question 4: Using Cursor Variables and Dynamic SQL

Task:
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.

PL/SQL Block:

```
DECLARE
  TYPE EmpCurType IS REF CURSOR;
  emp_cur EmpCurType;
  emp_record Employees%ROWTYPE;
  salary_threshold NUMBER := 50000; -- Example threshold

BEGIN
```

```
  OPEN emp_cur FOR 'SELECT EmployeeID, FirstName, LastName FROM Employees
WHERE Salary > ' || salary_threshold;

  LOOP
    FETCH emp_cur INTO emp_record;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('EmployeeID: ' || emp_record.EmployeeID || ', Name: ' ||
emp_record.FirstName || ' ' || emp_record.LastName);
  END LOOP;

  CLOSE emp_cur;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An error occurred: ' || SQLERRM);
END;
```

Explanation of Dynamic SQL:

- Dynamic SQL is constructed at runtime, allowing for flexibility in building SQL statements based on variable conditions. In this block, a cursor variable (`EmpCurType`) is used with dynamic SQL to fetch records from the `Employees` table where the salary exceeds a specified threshold.

Question 5: Designing Pipelined Function for Sales Data

Task:

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.

PL/SQL Block:

```
CREATE OR REPLACE TYPE SalesRecord IS OBJECT (
  OrderID NUMBER,
  CustomerID NUMBER,
  OrderAmount NUMBER
);

CREATE OR REPLACE TYPE SalesTable IS TABLE OF SalesRecord;

CREATE OR REPLACE FUNCTION get_sales_data(p_month IN NUMBER, p_year IN
NUMBER)
RETURN SalesTable PIPELINED
IS
BEGIN
  FOR rec IN (
    SELECT OrderID, CustomerID, OrderAmount
    FROM Sales
    WHERE EXTRACT(MONTH FROM OrderDate) = p_month
    AND EXTRACT(YEAR FROM OrderDate) = p_year
  ) LOOP
    PIPE ROW(SalesRecord(rec.OrderID, rec.CustomerID, rec.OrderAmount));
  END LOOP;

  RETURN;
END;
```

Explanation of Pipelined Table Functions:

- Pipelined functions allow for row-by-row processing and immediate returning of rows to the client as they are produced. This reduces memory consumption and improves response

times for large datasets, as rows are processed and sent incrementally rather than in a single batch.

Each solution is crafted with a focus on clarity, efficiency, and handling edge cases, ensuring that the PL/SQL code is robust and maintainable.