ASSIGNMENT – 3

**PL/SQL Questions**

**Question 1: Handling Division Operation**

**TABLE:**

|  |  |  |
| --- | --- | --- |
| ID | DIVIDEND | DIVISOR |
| 1 | 100 | 25 |
| 2 | 50 | 0 |
| 3 | 75 | 15 |

**TABLE QUERY:**

CREATE TABLE division\_input (

id NUMBER PRIMARY KEY,

dividend NUMBER NOT NULL,

divisor NUMBER NOT NULL

);

INSERT INTO division\_input (id, dividend, divisor) VALUES (1, 100, 25);

INSERT INTO division\_input (id, dividend, divisor) VALUES (2, 50, 0); -- This will cause ZERO\_DIVIDE

INSERT INTO division\_input (id, dividend, divisor) VALUES (3, 75, 15);

**DELIVERABLES:**

1. **PL/SQL BLOCK:**

DECLARE

v\_dividend NUMBER;

v\_divisor NUMBER;

v\_result NUMBER;

BEGIN

FOR rec IN (SELECT id, dividend, divisor FROM division\_input) LOOP

BEGIN

-- Fetch the dividend and divisor

v\_dividend := rec.dividend;

v\_divisor := rec.divisor;

-- Perform the division

v\_result := v\_dividend / v\_divisor;

-- Output the result

DBMS\_OUTPUT.PUT\_LINE('ID: ' || rec.id || ' - Result: ' || v\_result);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

-- Handle division by zero

DBMS\_OUTPUT.PUT\_LINE('ID: ' || rec.id || ' - Error: Division by zero is not allowed.');

END;

END LOOP;

END;

**2.EXPLANATION:**

 **Cursor Loop:** The PL/SQL block iterates over each record in the division\_input table using a cursor loop (FOR rec IN ... LOOP). This allows processing of each row individually.

 **Division Operation:**

* The values of dividend and divisor are fetched into variables v\_dividend and v\_divisor.
* The division operation v\_result := v\_dividend / v\_divisor is performed.

 **Exception Handling (ZERO\_DIVIDE):**

* The ZERO\_DIVIDE exception is caught in the EXCEPTION block. If a division by zero occurs (e.g., for ID = 2), the block prints an appropriate error message for that specific ID.

 **Output:**

* If the division is successful, the result is displayed using DBMS\_OUTPUT.PUT\_LINE.
* If an error occurs, an error message is displayed instead, indicating which record caused the issue.

**OUTPUT TABLE:**

|  |  |
| --- | --- |
| ID | RESULT |
| 1 | 4 |
| 2 | 0 NOT ALLOWED |
| 3 | 5 |

**Question 2: Updating Rows with FORALL**

**TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| Employee id | First name | Last name | salary |
| 1 | John | Doe | 50000 |
| 2 | Jahn | Smith | 55000 |
| 3 | Emiley | Jones | 65000 |
| 4 | Michelle | Brown | 60000 |
| 5 | laura | devis | 75000 |

**TABLE QUERY:**

CREATE TABLE Employees (

employee\_id NUMBER PRIMARY KEY,

first\_name VARCHAR2(50) NOT NULL,

last\_name VARCHAR2(50) NOT NULL,

salary NUMBER NOT NULL

);

INSERT INTO Employees (employee\_id, first\_name, last\_name, salary) VALUES (1, 'John', 'Doe', 50000);

INSERT INTO Employees (employee\_id, first\_name, last\_name, salary) VALUES (2, 'Jane', 'Smith', 55000);

INSERT INTO Employees (employee\_id, first\_name, last\_name, salary) VALUES (3, 'Emily', 'Jones', 60000);

INSERT INTO Employees (employee\_id, first\_name, last\_name, salary) VALUES (4, 'Michael', 'Brown', 65000);

INSERT INTO Employees (employee\_id, first\_name, last\_name, salary) VALUES (5, 'Laura', 'Davis', 70000);

**DELIVERABLES:**

* 1. **PL/SQL BLOCK:**

DECLARE

TYPE t\_emp\_id IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

TYPE t\_salary\_inc IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

v\_emp\_ids t\_emp\_id;

v\_salary\_incs t\_salary\_inc;

BEGIN

-- Initialize the arrays with employee IDs and salary increments

v\_emp\_ids(1) := 1; -- Employee ID 1

v\_emp\_ids(2) := 2; -- Employee ID 2

v\_emp\_ids(3) := 3; -- Employee ID 3

v\_emp\_ids(4) := 4; -- Employee ID 4

v\_emp\_ids(5) := 5; -- Employee ID 5

v\_salary\_incs(1) := 5000; -- Increase salary by 5000 for Employee ID 1

v\_salary\_incs(2) := 3000; -- Increase salary by 3000 for Employee ID 2

v\_salary\_incs(3) := 4000; -- Increase salary by 4000 for Employee ID 3

v\_salary\_incs(4) := 2500; -- Increase salary by 2500 for Employee ID 4

v\_salary\_incs(5) := 3500; -- Increase salary by 3500 for Employee ID 5

-- Use FORALL to efficiently update salaries

FORALL i IN v\_emp\_ids.FIRST .. v\_emp\_ids.LAST

UPDATE Employees

SET salary = salary + v\_salary\_incs(i)

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

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

END;

/

**2.EXPLANATION:**

 **PL/SQL Collections:**

* **t\_emp\_id**: A collection (associative array) to store the employee IDs.
* **t\_salary\_inc**: A collection (associative array) to store the corresponding salary increments for each employee.

 **Initialization of Arrays:**

* The arrays v\_emp\_ids and v\_salary\_incs are populated with employee IDs and their corresponding salary increments.

 **FORALL Statement:**

* The FORALL statement is used to efficiently iterate over the arrays and update the salaries in the Employees table in a single context switch, reducing the overhead associated with multiple UPDATE statements.

 **UPDATE Statement:**

* The UPDATE statement within FORALL updates the salary column by adding the corresponding increment (v\_salary\_incs(i)) to the current salary for each employee.

**OUTPUT TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| EMPLOYEE ID | FIRST NAME | LAST NAME | SALARY(UPDATED) |
| 1 | John | Doe | 55000 |
| 2 | Jahn | Smith | 58000 |
| 3 | Emiley | Jones | 69000 |
| 4 | Michelle | Brown | 62500 |
| 5 | laura | devis | 78500 |

**Question 3: Implementing Nested Table Procedure**

**TABLE:**

**DEPARTMENT TABLE:**

|  |  |
| --- | --- |
| DEPARTMENT ID | DEPARTMENT NAME |
| 1 | IT |
| 2 | SALES |
| 3 | HR |

**EMPLOYEE TABLE:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| EMPLOYEE ID | FIRST NAME | LAST NAME | SALARY | DEPARTMENT ID |
| 1 | John | Doe | 50000 | 1 |
| 2 | Jahn | Smith | 55000 | 2 |
| 3 | Emiley | Jones | 65000 | 3 |
| 4 | Michelle | Brown | 60000 | 1 |
| 5 | laura | devis | 75000 | 2 |

**TABLE QUERY:**

CREATE TABLE Departments (

department\_id NUMBER PRIMARY KEY,

department\_name VARCHAR2(50) NOT NULL

);

CREATE TABLE Employees (

employee\_id NUMBER PRIMARY KEY,

first\_name VARCHAR2(50) NOT NULL,

last\_name VARCHAR2(50) NOT NULL,

salary NUMBER NOT NULL,

department\_id NUMBER,

FOREIGN KEY (department\_id) REFERENCES Departments(department\_id)

);

**DELIVERABLES:**

* 1. **PL/SQL BLOCK:**

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

p\_department\_id IN NUMBER,

p\_employees OUT SYS\_REFCURSOR

) IS

BEGIN

-- Open a cursor to return the employees belonging to the specified department

OPEN p\_employees FOR

SELECT employee\_id, first\_name, last\_name, salary

FROM Employees

WHERE department\_id = p\_department\_id;

END;

/

* 1. **EXPLANATION:**

**Input Parameter (p\_department\_id):**

Accepts the department ID as input to filter employees by their department.

**Output Parameter (p\_employees):**

A SYS\_REFCURSOR type is used to return the result set (a collection of employees) that matches the department ID.

**Cursor for Employee Retrieval:**

The procedure opens a cursor (OPEN p\_employees FOR ...) that retrieves all employees belonging to the specified department.

**OUTPUT TABLE:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| | Employee ID | | --- | | FIRST NAME | LAST NAME | SALARY |
| 1 | JOHN | DAE | 50000 |
| 4 | MICHELLE | BROWN | 60000 |
| |  | | --- | | 2 | | JAHN | SMITH | 55000 |
| 5 | LAURA | DEVIS | 75000 |
| 3 | EMILEY | JANES | 65000 |

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

**TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| EMPLOYEE ID | FIRST NAME | LAST NAME | SALARY |
| 1 | John | Doe | 50000 |
| 2 | Jahn | Smith | 55000 |
| 3 | Emiley | Jones | 65000 |
| 4 | Michelle | Brown | 60000 |
| 5 | Emiley | Jones | 75000 |

**TABLE QUERY:**

CREATE TABLE Employees (

employee\_id NUMBER PRIMARY KEY,

first\_name VARCHAR2(50) NOT NULL,

last\_name VARCHAR2(50) NOT NULL,

salary NUMBER NOT NULL

);

**DELIVERABLES:**

* 1. **PL/SQL BLOCK:**

DECLARE

TYPE ref\_cursor IS REF CURSOR;

c\_employees ref\_cursor;

v\_employee\_id Employees.employee\_id%TYPE;

v\_first\_name Employees.first\_name%TYPE;

v\_last\_name Employees.last\_name%TYPE;

v\_salary\_threshold NUMBER := 60000;

v\_sql VARCHAR2(200);

BEGIN

-- Construct the dynamic SQL query

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

-- Open the cursor with the dynamic SQL

OPEN c\_employees FOR v\_sql USING v\_salary\_threshold;

-- Fetch and display the results

LOOP

FETCH c\_employees INTO v\_employee\_id, v\_first\_name, v\_last\_name;

EXIT WHEN c\_employees%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('ID: ' || v\_employee\_id || ', Name: ' || v\_first\_name || ' ' || v\_last\_name);

END LOOP;

-- Close the cursor

CLOSE c\_employees;

END;

/

* 1. **EXPLANATION:**

1. **Ref Cursor Type (ref\_cursor):**
   * A REF CURSOR type is defined to create a cursor variable that can be associated with different queries dynamically.
2. **Cursor Variable (c\_employees):**
   * A cursor variable is declared using the ref\_cursor type. This cursor will be used to execute the dynamic SQL query.
3. **Dynamic SQL (v\_sql):**
   * The dynamic SQL statement is constructed in the v\_sql variable. It selects employee\_id, first\_name, and last\_name for employees whose salary is greater than the specified threshold.
4. **Open Cursor with Dynamic SQL:**
   * The cursor is opened with the dynamic SQL using the OPEN FOR statement, and the salary threshold is passed using a bind variable.
5. **Fetching and Displaying Results:**
   * A loop is used to fetch each row from the cursor and display the employee details. The loop exits when there are no more rows to fetch.
6. **Closing the Cursor:**
   * Finally, the cursor is closed to release resources.

**OUTPUT TABLE:**

|  |  |  |
| --- | --- | --- |
| **EMPLOYEE ID** | **FIRST NAME** | **LAST NAME** |
| **3** | Emiley | Jones |
| **5** | LAURA | DEVIS |

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

**TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| ORDER ID | CUSTOMER ID | ORDER DATE | ORDER AMOUNT |
| 1 | 101 | 05-07-2024 | 150 |
| 2 | 102 | 12-07-2024 | 250 |
| 3 | 103 | 10-08-2024 | 200 |
| 4 | 104 | 15-08-2024 | 300 |
| 5 | 105 | 20-08-2024 | 350 |

**TABLE QUERY:**

CREATE TABLE Sales (

OrderID NUMBER PRIMARY KEY,

CustomerID NUMBER NOT NULL,

OrderDate DATE NOT NULL,

OrderAmount NUMBER NOT NULL

);

**DELIVERABLES:**

* 1. **PL/SQL BLOCK:**

CREATE OR REPLACE TYPE sales\_record AS OBJECT (

OrderID NUMBER,

CustomerID NUMBER,

OrderAmount NUMBER

);

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

CREATE OR REPLACE FUNCTION get\_sales\_data (

p\_month IN NUMBER,

p\_year IN NUMBER

) RETURN sales\_table PIPELINED

IS

v\_sales\_row sales\_record;

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

v\_sales\_row := sales\_record(rec.OrderID, rec.CustomerID, rec.OrderAmount);

PIPE ROW(v\_sales\_row);

END LOOP;

RETURN;

END;

/

* 1. **EXPLANATION:**

 **User-Defined Types:**

* **sales\_record:** An object type representing a single row in the output, containing OrderID, CustomerID, and OrderAmount.
* **sales\_table:** A table type (nested table) of sales\_record objects, representing the collection returned by the function.

 **Function Logic:**

* The function accepts p\_month and p\_year as input parameters to filter orders based on the specified month and year.
* The FOR loop iterates over the result set of the filtered query.
* The PIPE ROW(v\_sales\_row) statement pipes each row of the result set to the calling query, making the function behave like a table.

**OUTPUT TABLE:**

|  |  |  |
| --- | --- | --- |
| ORDER ID | CUSTOMER ID | ORDER AMOUNT |
| 3 | 103 | 300 |
| 4 | 104 | 400 |
| 5 | 105 | 350 |