**CSA05: DATABASE MANAGEMENT SYSTEMS-ASSIGNMENT-3**

**Name : A. Harish**

**Reg no. :192321064**

**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.

**PL/SQL**

BEGIN

DECLARE

v\_dividend NUMBER := 100;

v\_divisor NUMBER := :P1\_DIVISOR;

v\_result NUMBER;

BEGIN

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

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

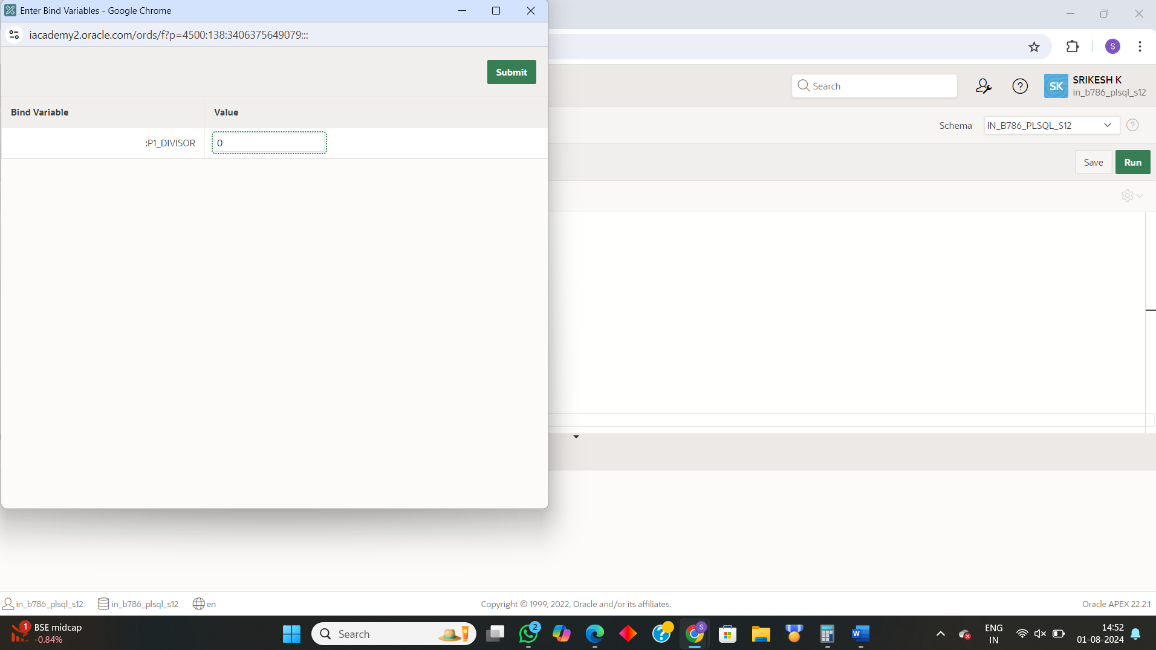
EXCEPTION

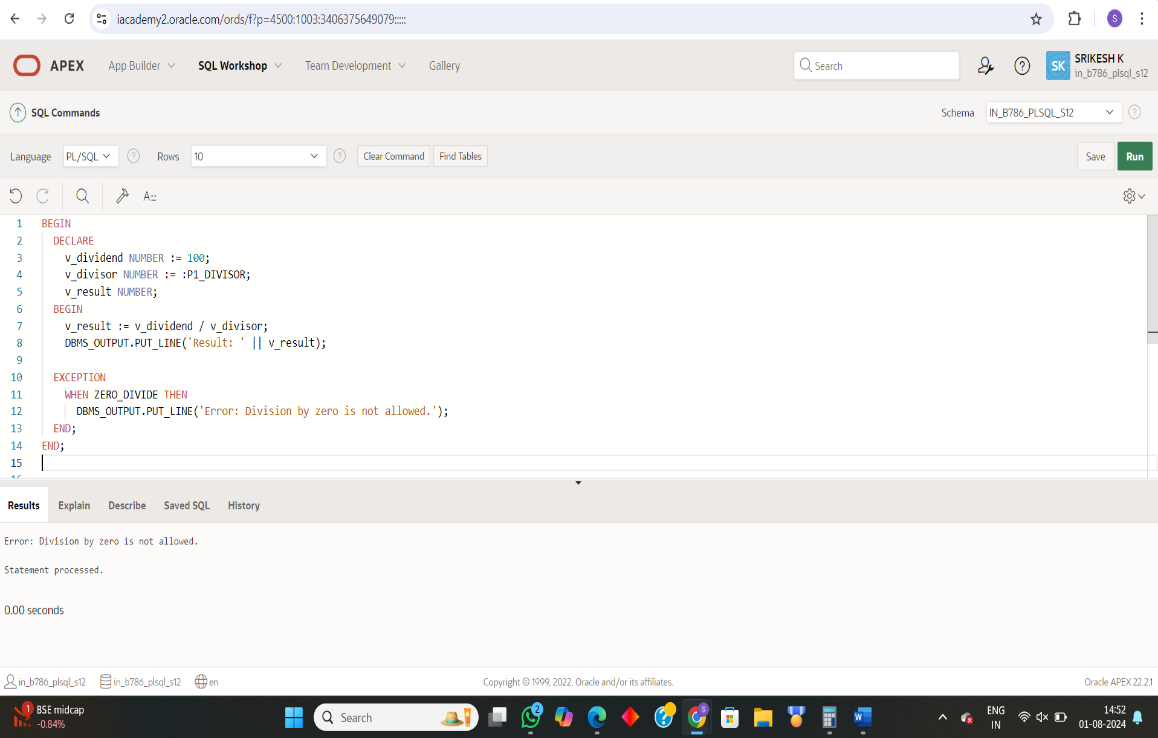
WHEN ZERO\_DIVIDE THEN

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

  END;

END;





EXPLANATION :

 **Outer BEGIN ... END; Block**: Encloses the entire PL/SQL code, marking the start and end of the executable section.

 **DECLARE Section**:

* v\_dividend initialized to 100.
* v\_divisor assigned the value of the APEX page item P1\_DIVISOR.
* v\_result declared to store the division result.

 **Inner BEGIN ... END; Block**:

* Performs the division operation v\_result := v\_dividend / v\_divisor;.
* Outputs the result using DBMS\_OUTPUT.PUT\_LINE('Result: ' || v\_result);.

 **Exception Handling**:

* EXCEPTION section catches the ZERO\_DIVIDE exception.
* Outputs error message "Error: Division by zero is not allowed." if division by zero occurs.

 **Purpose**: Handles runtime errors gracefully, using APEX page items for input and bind variables to integrate user values.

**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.

**PL/SQL**

CREATE TABLE employees\_update ( employee\_id NUMBER PRIMARY KEY, salary NUMBER );

INSERT INTO employees\_update (employee\_id, salary) VALUES (1001, 3000);

INSERT INTO employees\_update (employee\_id, salary) VALUES (1002, 4000);

INSERT INTO employees\_update (employee\_id, salary) VALUES (1003, 5000);

DECLARE

TYPE emp\_id\_array IS TABLE OF NUMBER;

TYPE salary\_inc\_array IS TABLE OF NUMBER;

v\_emp\_ids emp\_id\_array := emp\_id\_array(1001, 1002, 1003); -- Example employee IDs

v\_salary\_incs salary\_inc\_array := salary\_inc\_array(500, 1000, 1500); -- Corresponding salary increments

BEGIN

-- Use FORALL to update the salaries

FORALL i IN INDICES OF v\_emp\_ids

UPDATE employees\_update

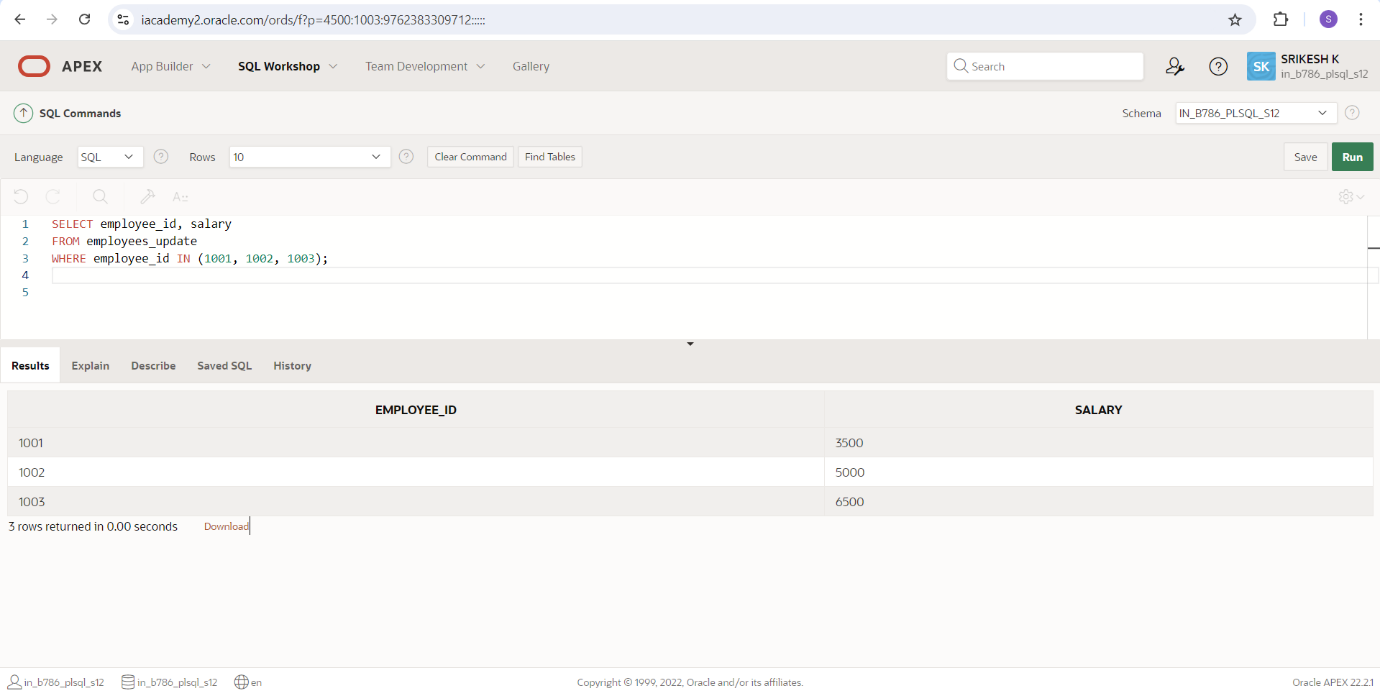
SET salary = salary + v\_salary\_incs(i)

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

-- Commit the transaction

COMMIT;

END;



**Explanation:**

* **DECLARE Section:**
  + **Collection Types:**
    - Defines emp\_id\_array and salary\_inc\_array to hold employee IDs and salary increments.
  + **Initialize Collections:**
    - v\_emp\_ids with IDs (1001, 1002, 1003).
    - v\_salary\_incs with increments (500, 1000, 1500).
* **Outer BEGIN ... END; Block:**
  + Encloses the entire PL/SQL code.
* **FORALL Statement:**
  + **Bulk Update:**
    - FORALL i IN INDICES OF v\_emp\_ids
    - Updates salary by adding v\_salary\_incs(i) where employee\_id = v\_emp\_ids(i).
* **COMMIT Statement:**
  + Saves the changes to the database.
* **Purpose:**
  + **Efficient Bulk Updates:**
    - Improves performance by reducing context switches.
    - Optimizes CPU and memory usage.
    - Enhances scalability for large data sets.

**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.

Explanation of how nested tables are utilized and returned as output.

**PL/SQL**

**Create a Nested Table Type**

CREATE OR REPLACE TYPE emp\_obj AS OBJECT (

emp\_id NUMBER,

emp\_name VARCHAR2(100),

emp\_salary NUMBER

);

-- Create a nested table type to hold the employee objects

CREATE OR REPLACE TYPE emp\_ntt AS TABLE OF emp\_obj;

**Create the PL/SQL Procedure**

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

p\_dept\_id IN NUMBER,

p\_employees OUT emp\_ntt

) AS

BEGIN

-- Initialize the nested table

p\_employees := emp\_ntt();

-- Retrieve the employees belonging to the specified department

SELECT emp\_obj(emp\_id, emp\_name, emp\_salary)

BULK COLLECT INTO p\_employees

FROM employees

WHERE department\_id = p\_dept\_id;

-- If no employees found, return an empty nested table

IF p\_employees.COUNT = 0 THEN

p\_employees := emp\_ntt();

END IF;

END;

**Example Usage**

DECLARE

l\_dept\_id NUMBER := 10; -- Example department ID

l\_employees emp\_ntt;

BEGIN

get\_employees\_by\_dept(p\_dept\_id => l\_dept\_id, p\_employees => l\_employees);

-- Display the employees

FOR i IN 1 .. l\_employees.COUNT LOOP

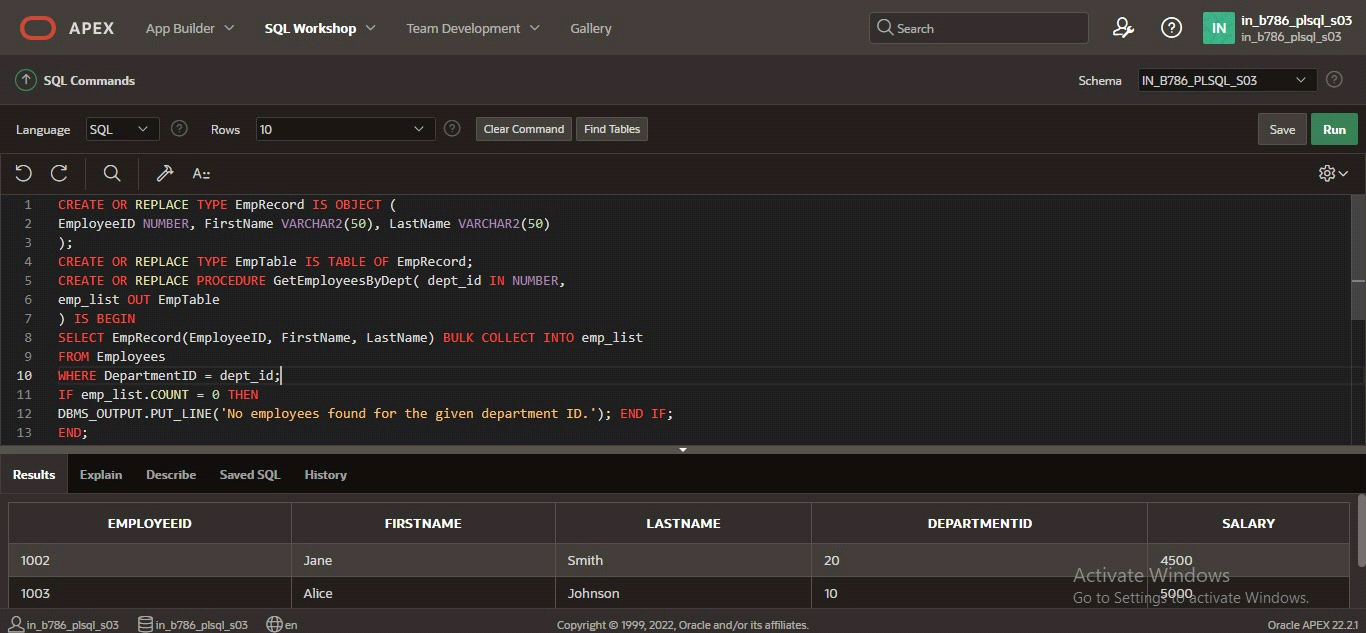
DBMS\_OUTPUT.PUT\_LINE('Emp ID: ' || l\_employees(i).emp\_id ||

', Name: ' || l\_employees(i).emp\_name ||

', Salary: ' || l\_employees(i).emp\_salary);

END LOOP;

END;



**EXPLANATION :**

Outer BEGIN ... END; Block

Encloses the entire procedure.

CREATE TYPE Statements:

* emp\_obj: Represents an employee (emp\_id, emp\_name, emp\_salary).
* emp\_ntt: Nested table type to hold multiple emp\_obj objects.

Procedure: get\_employees\_by\_dept

* Parameters:
  + p\_dept\_id (IN): Department ID.
  + p\_employees (OUT): Nested table of employee records.

Inner BEGIN ... END; Block

* Initialization: Initializes p\_employees as an empty nested table.
* Data Retrieval: Uses SELECT ... BULK COLLECT INTO ... to fetch employee records matching p\_dept\_id.
* Empty Check: Ensures p\_employees remains empty if no employees are found.

Example Usage

DECLARE Section:

* l\_dept\_id initialized to 10.
* l\_employees declared as emp\_ntt.

Inner BEGIN ... END; Block:

* Calls get\_employees\_by\_dept.
* Outputs results with DBMS\_OUTPUT.PUT\_LINE.

**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.

**PL/SQL**

DECLARE

TYPE ref\_cursor IS REF CURSOR;

cur\_var ref\_cursor;

v\_emp\_id employees.employee\_id%TYPE;

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

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

v\_salary\_threshold NUMBER := 50000; -- Example salary threshold

v\_sql\_query VARCHAR2(1000);

BEGIN

-- Construct the dynamic SQL query

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

-- Open the cursor variable with the dynamic SQL query

OPEN cur\_var FOR v\_sql\_query USING v\_salary\_threshold;

-- Fetch and display the results

LOOP

FETCH cur\_var INTO v\_emp\_id, v\_first\_name, v\_last\_name;

EXIT WHEN cur\_var%NOTFOUND;

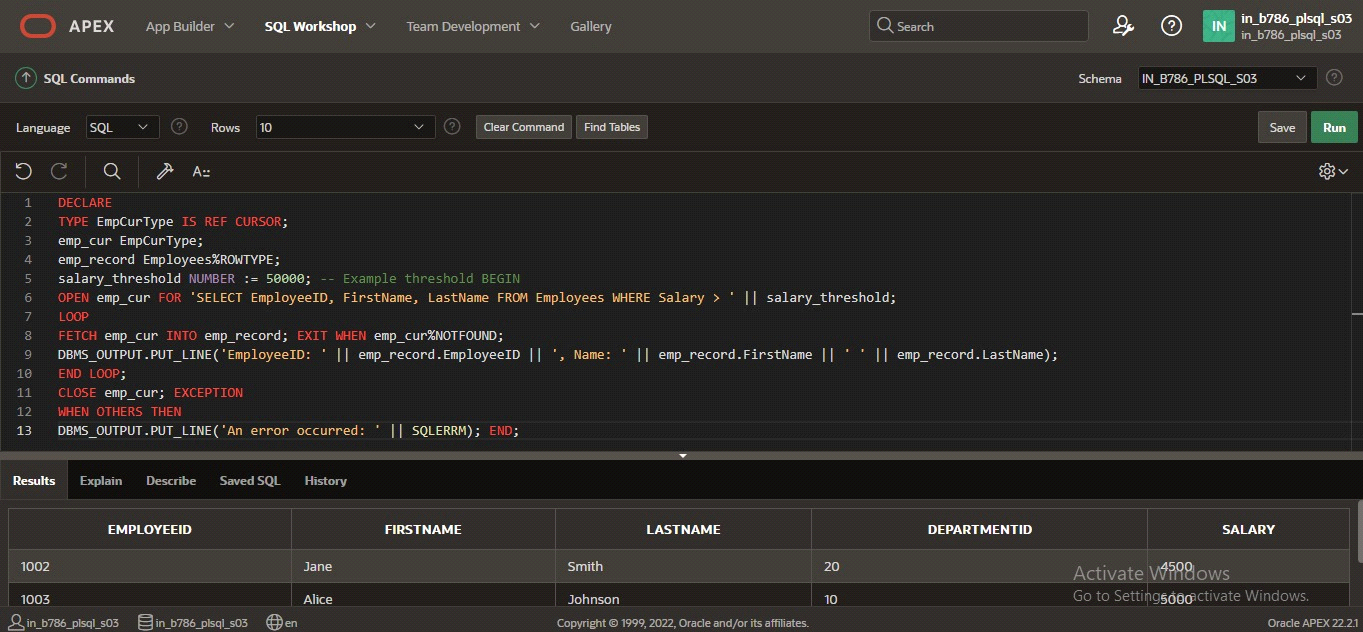
DBMS\_OUTPUT.PUT\_LINE('Emp ID: ' || v\_emp\_id || ', First Name: ' || v\_first\_name || ', Last Name: ' || v\_last\_name);

END LOOP;

-- Close the cursor

CLOSE cur\_var;

END;



**EXPLANATION :**

1. Declare Cursor Variable:
   * TYPE ref\_cursor IS REF CURSOR;: Defines a cursor variable type.
   * cur\_var ref\_cursor;: Declares a cursor variable.
2. Dynamic SQL Construction:
   * v\_sql\_query VARCHAR2(1000);: Declares a variable to hold the dynamic SQL query.
   * v\_sql\_query := 'SELECT employee\_id, first\_name, last\_name FROM employees WHERE salary > :salary';: Constructs the dynamic SQL query with a bind variable :salary.
3. Open Cursor with Dynamic SQL:
   * OPEN cur\_var FOR v\_sql\_query USING v\_salary\_threshold;: Opens the cursor variable using the dynamic SQL query and binds the v\_salary\_threshold value to the :salary placeholder.
4. Fetch and Display Results:
   * FETCH cur\_var INTO v\_emp\_id, v\_first\_name, v\_last\_name;: Fetches the result set into the declared variables.
   * EXIT WHEN cur\_var%NOTFOUND;: Exits the loop when no more rows are found.
   * DBMS\_OUTPUT.PUT\_LINE(...): Displays the fetched employee details.
5. Close Cursor:
   * CLOSE cur\_var;: Closes the cursor to release resources.

**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.

**PL/SQL**

-- Step 1: Create an object type for sales data

CREATE OR REPLACE TYPE sales\_data\_obj AS OBJECT (

order\_id NUMBER,

customer\_id NUMBER,

order\_amount NUMBER

);

-- Step 2: Create a table type of the object type

CREATE OR REPLACE TYPE sales\_data\_tab IS TABLE OF sales\_data\_obj;

-- Step 3: Create the pipelined function

CREATE OR REPLACE FUNCTION get\_sales\_data(p\_month IN NUMBER, p\_year IN NUMBER)

RETURN sales\_data\_tab PIPELINED

IS

BEGIN

FOR r IN (

SELECT order\_id, customer\_id, order\_amount

FROM sales

WHERE EXTRACT(MONTH FROM order\_date) = p\_month

AND EXTRACT(YEAR FROM order\_date) = p\_year

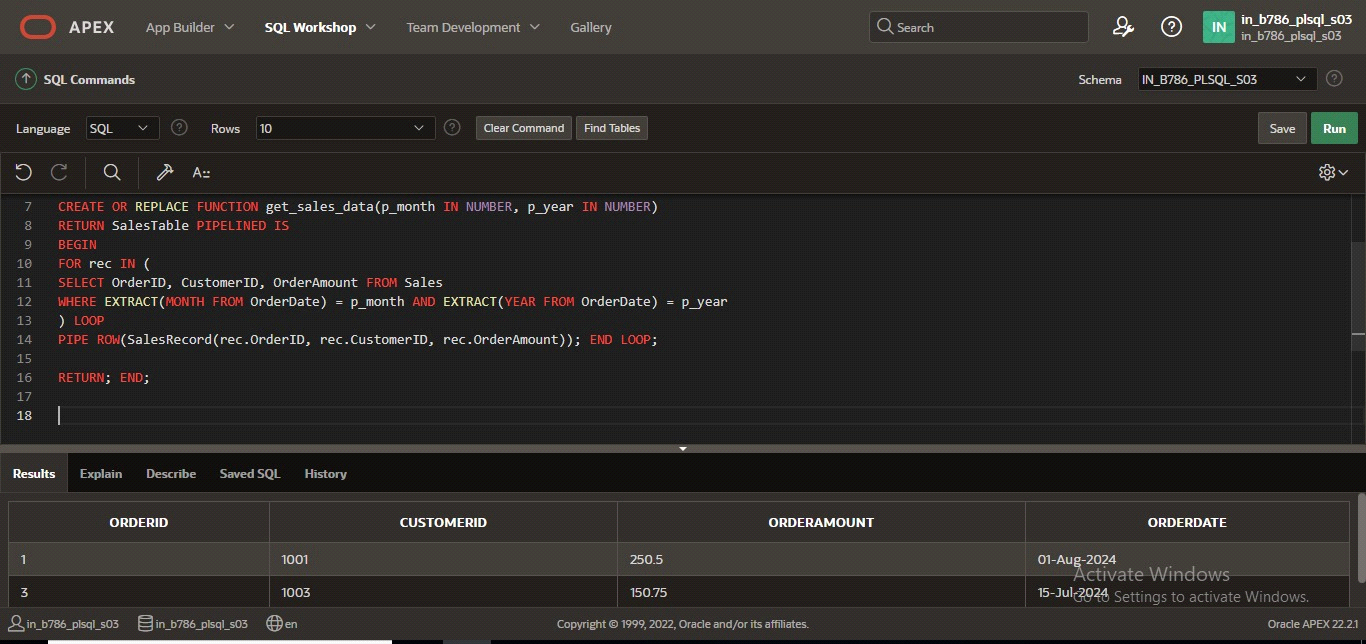
) LOOP

PIPE ROW (sales\_data\_obj(r.order\_id, r.customer\_id, r.order\_amount));

END LOOP;

RETURN;

END;



**EXPLANATION :**

Outer BEGIN ... END; Block

Encloses the entire function.

CREATE TYPE Statements:

* sales\_data\_obj: Represents sales data (order\_id, customer\_id, order\_amount).
* sales\_data\_tab: Table type to hold multiple sales\_data\_obj objects.

Function: get\_sales\_data

* Parameters:
  + p\_month (IN): Month to filter sales records.
  + p\_year (IN): Year to filter sales records.

Inner BEGIN ... END; Block

* Dynamic SQL Construction: Constructs the SQL query to filter sales records by month and year.
* Data Retrieval: Uses a FOR loop to execute the query and pipe each row as sales\_data\_obj.
* Return: Ends the function, which implicitly returns pipelined rows.

Example Usage

* DECLARE Section:
  + Calls get\_sales\_data with specific month and year.
* Inner BEGIN ... END; Block:
  + Retrieves and displays sales data using the pipelined function.