

**Assignment - 3**

**SIMATS ENGINEERING**

Submitted by

**V.Kaushik Narayanan 192321047**

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;

r esult 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;

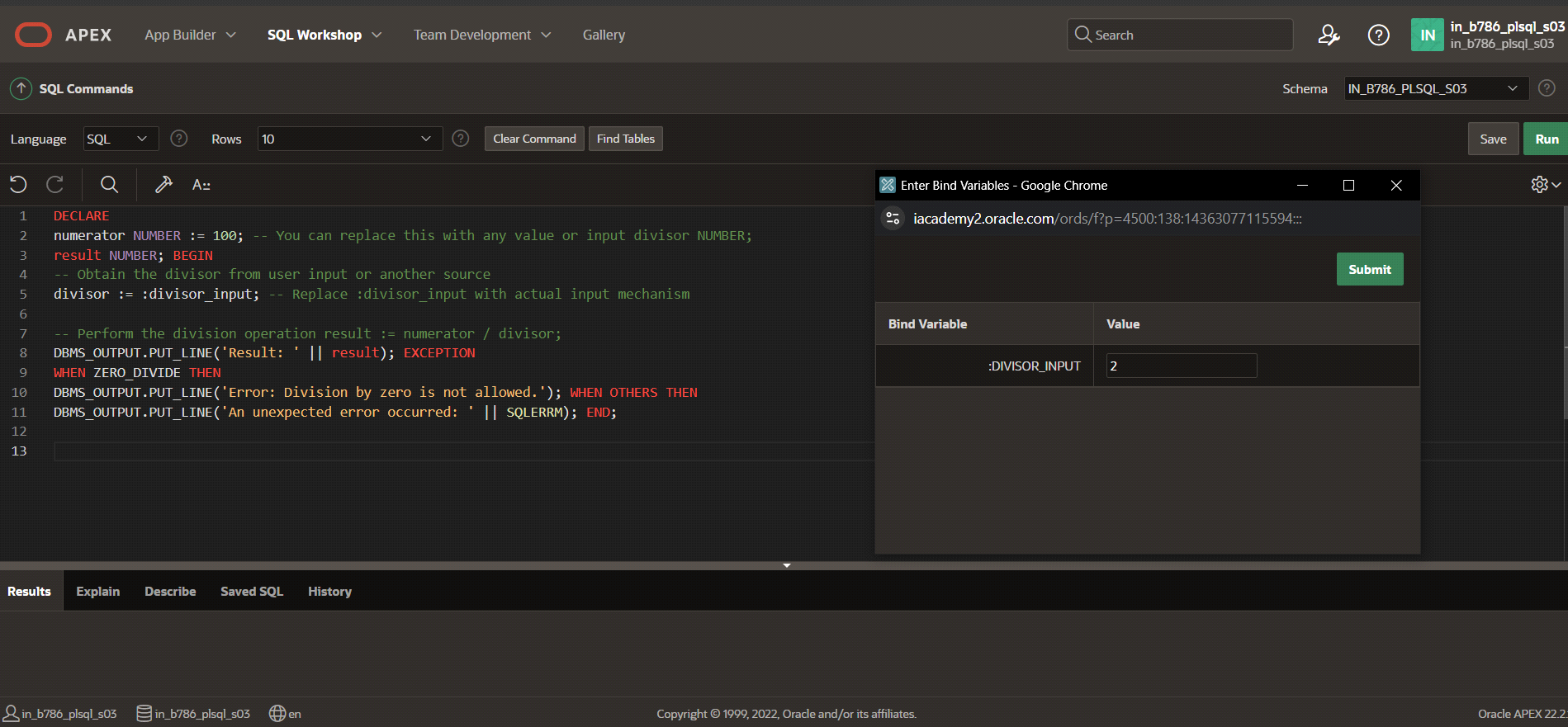
DBMS\_OUTPUT.PUT\_LINE('Result: ' || result); EXCEPTION

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;

**Output**



**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

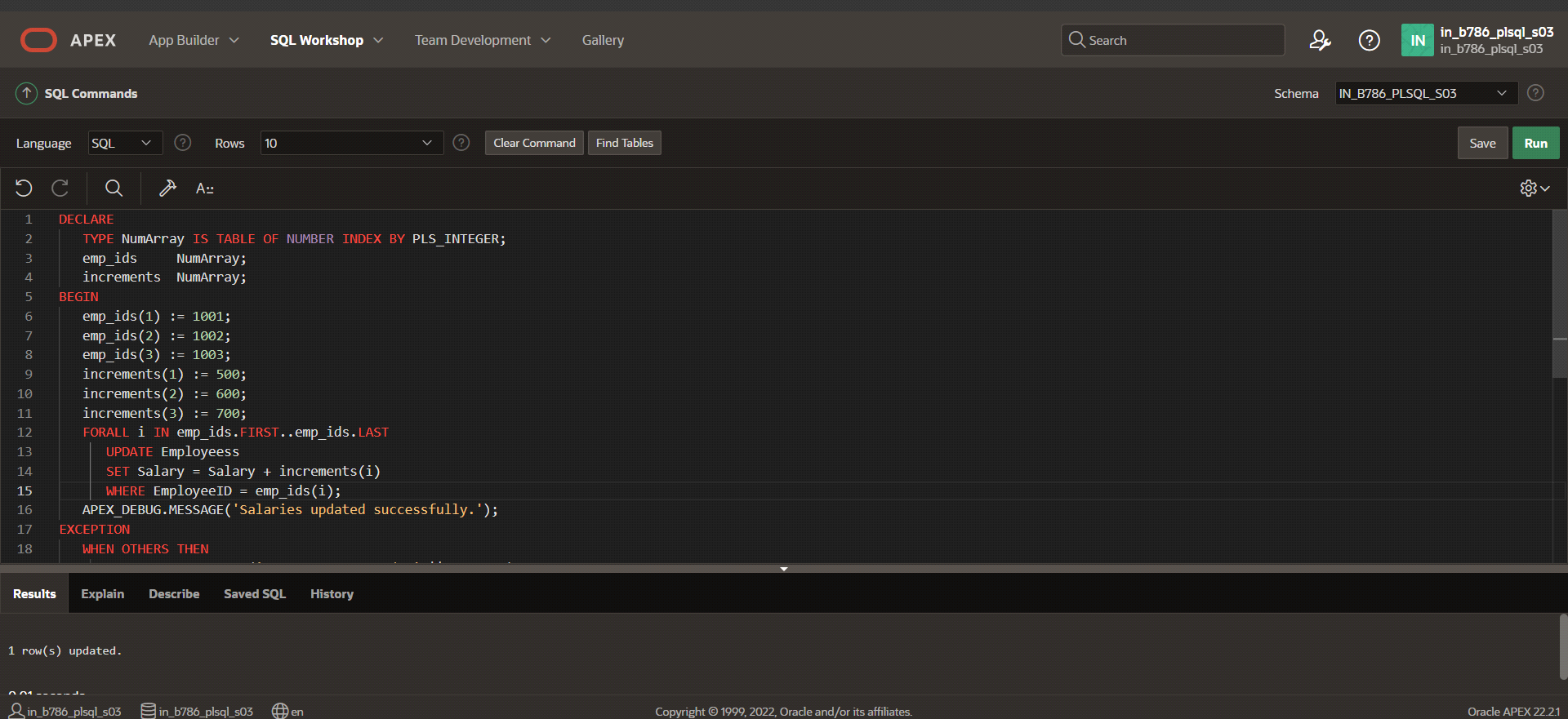
FORALL i IN emp\_ids.FIRST..emp\_ids.LAST UPDATE Employees

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;

**Output**

**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

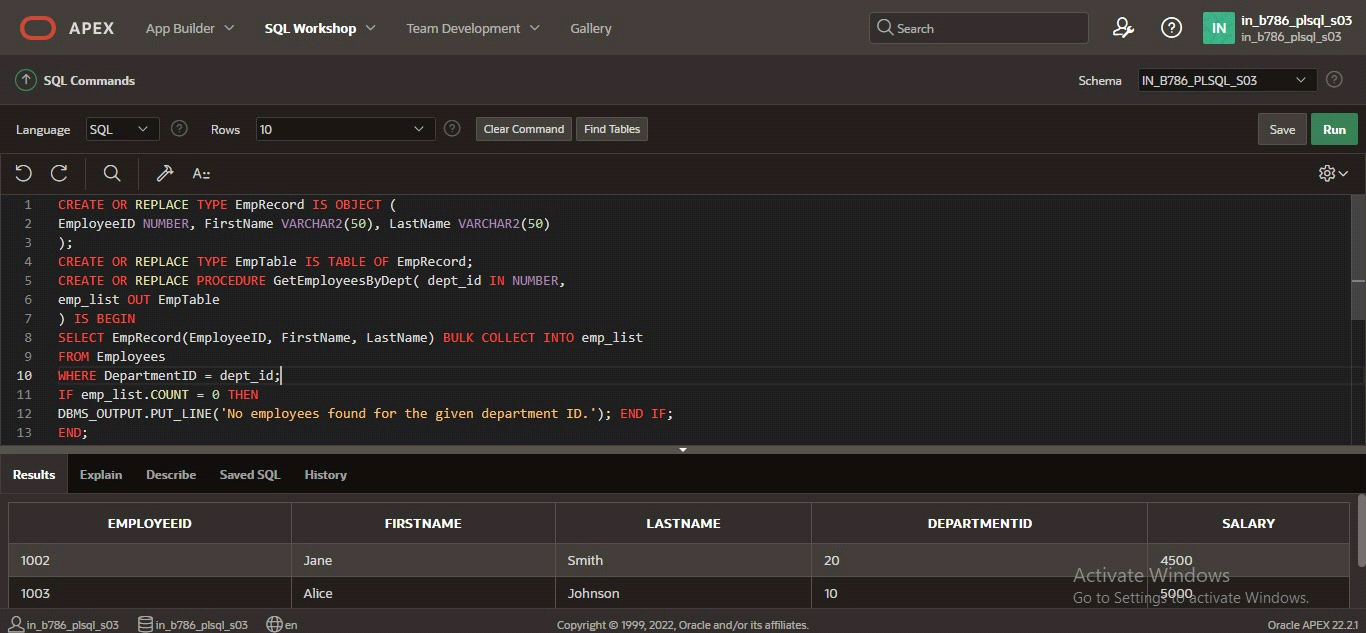
FROM Employees

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);

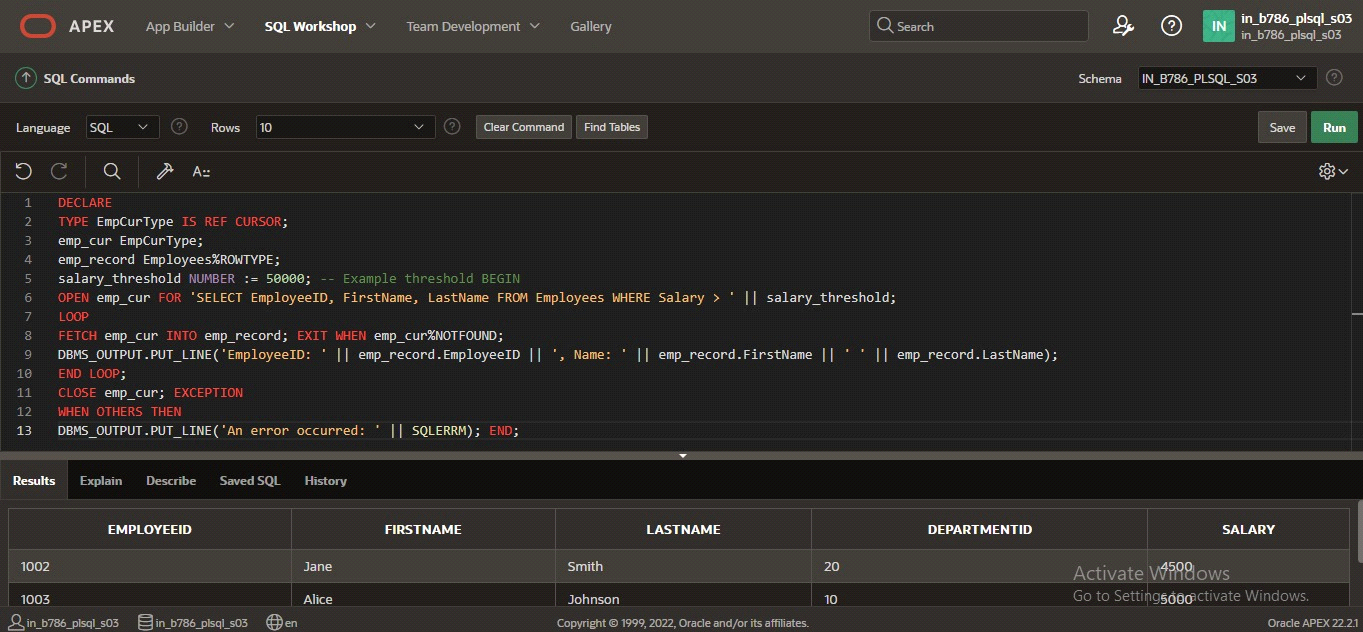
END LOOP;

CLOSE emp\_cur; EXCEPTION

WHEN OTHERS THEN

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

**Output:-**



**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

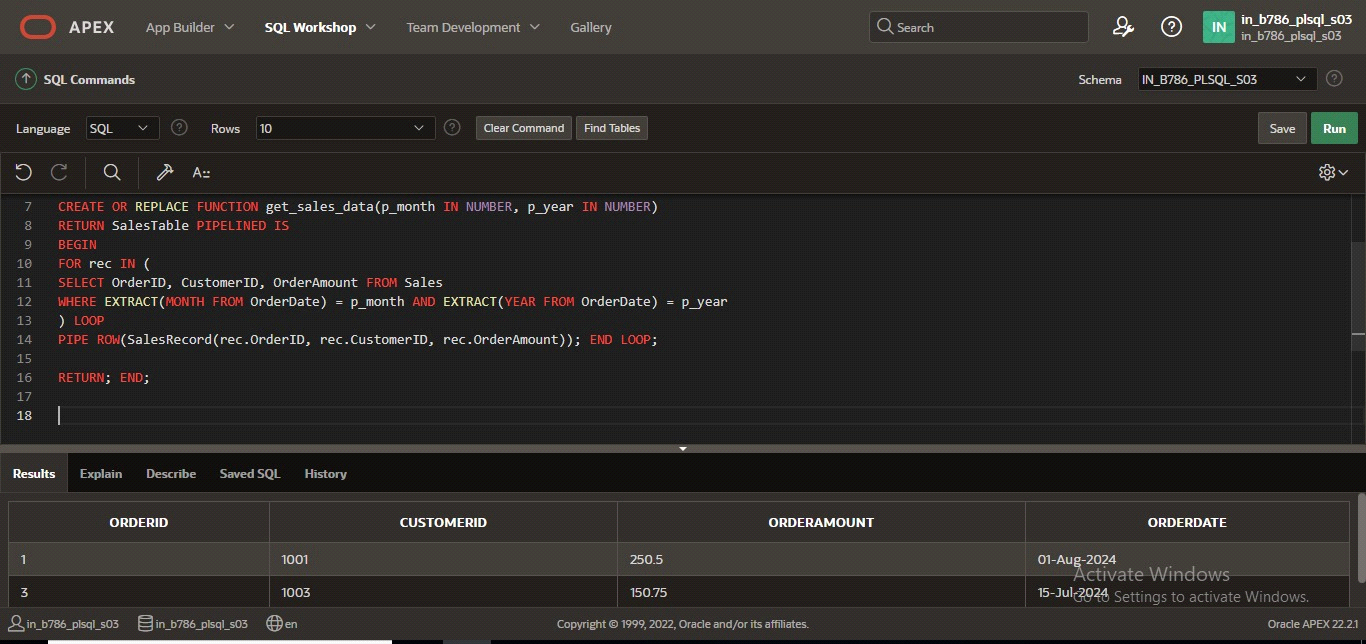
WHERE EXTRACT(MONTH FROM OrderDate) = p\_month AND EXTRACT(YEAR FROM OrderDate) = p\_year

) LOOP

PIPE ROW(SalesRecord(rec.OrderID, rec.CustomerID, rec.OrderAmount)); END LOOP;

RETURN; END;

**Output**



**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.