

# Assignment - 3

## SAVEETHA SCHOOL OF ENGINEERING

Submitted by

**Madhu Balan I (192321142)**

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

dividend NUMBER := 50; -- Example dividend

divisor NUMBER := 0;

result NUMBER;

BEGIN

-- Perform division

result := dividend / divisor;

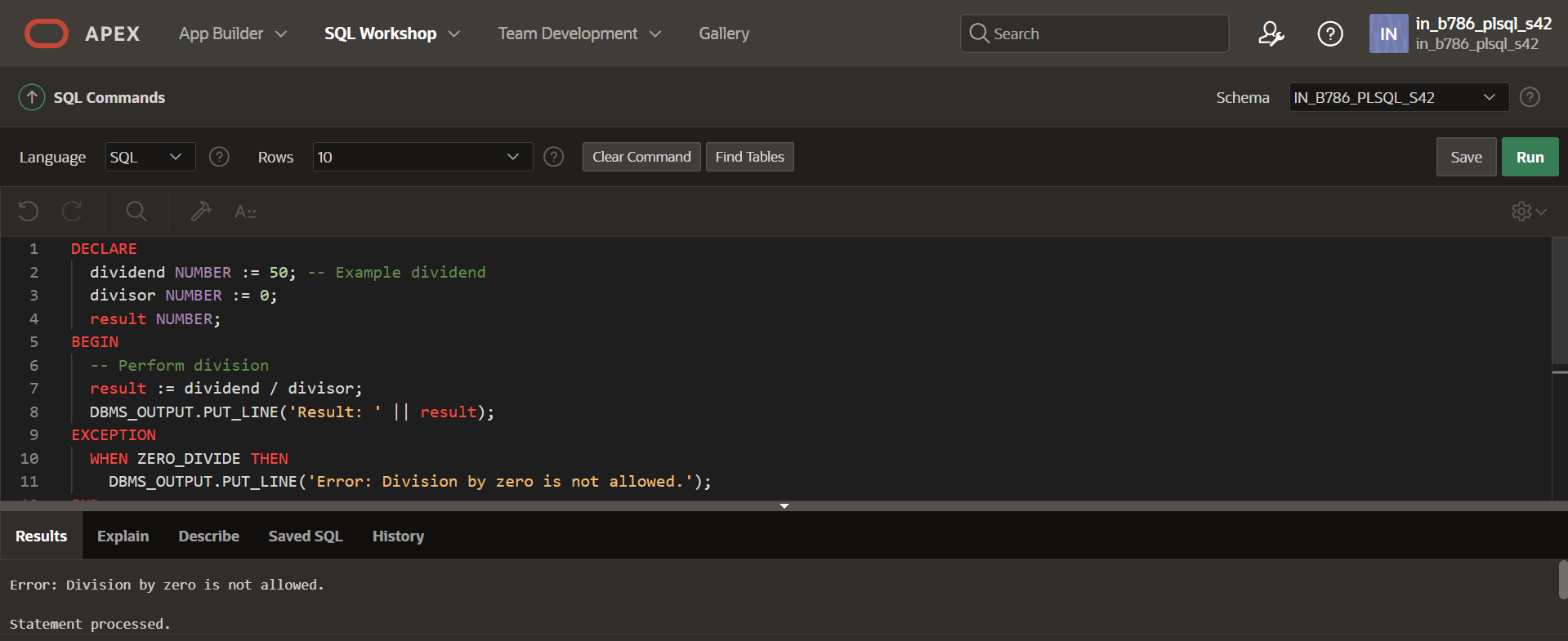
DBMS\_OUTPUT.PUT\_LINE('Result: ' || result);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

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

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 issue

## 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 emp\_id\_array IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

TYPE salary\_increment\_array IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

emp\_ids emp\_id\_array;

salary\_increments salary\_increment\_array;

BEGIN

emp\_ids(1) := 101;

emp\_ids(2) := 102;

emp\_ids(3) := 103;

salary\_increments(1) := 500;

salary\_increments(2) := 1000;

salary\_increments(3) := 750;

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

UPDATE Employees

SET salary = salary + salary\_increments(i)

WHERE employee\_id = emp\_ids(i);

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

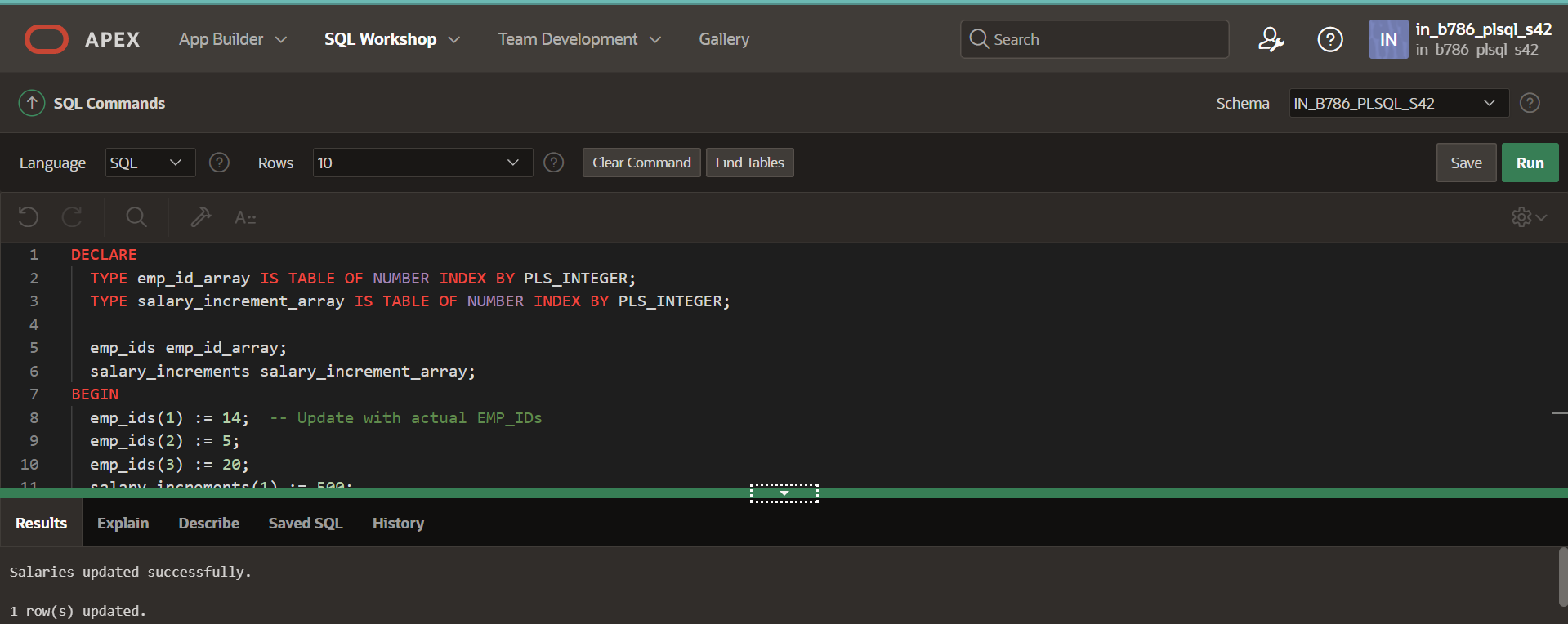
EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

END;

**Output**



## Description of How FORALL Improves Performanc

* 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 EmployeeRecord AS OBJECT (

## employee\_id NUMBER,

## first\_name VARCHAR2(50),

## last\_name VARCHAR2(50),

## salary NUMBER

## );

## CREATE OR REPLACE TYPE EmployeeTable AS TABLE OF EmployeeRecord;

## -- Step 2: Create the procedure to retrieve employees by department ID

## CREATE OR REPLACE PROCEDURE GetEmployeesByDepartment (

## p\_department\_id IN NUMBER,

## p\_employees OUT EmployeeTable

## ) AS

## BEGIN

## -- Initialize the nested table

## p\_employees := EmployeeTable();

## -- Retrieve employees belonging to the specified department

## SELECT EmployeeRecord(employee\_id, first\_name, last\_name, salary)

## BULK COLLECT INTO p\_employees

## FROM Employees

## WHERE department\_id = p\_department\_id;

## -- Check if any employees were found

## IF p\_employees.COUNT = 0 THEN

## DBMS\_OUTPUT.PUT\_LINE('No employees found for department ID ' || p\_department\_id);

## END IF;

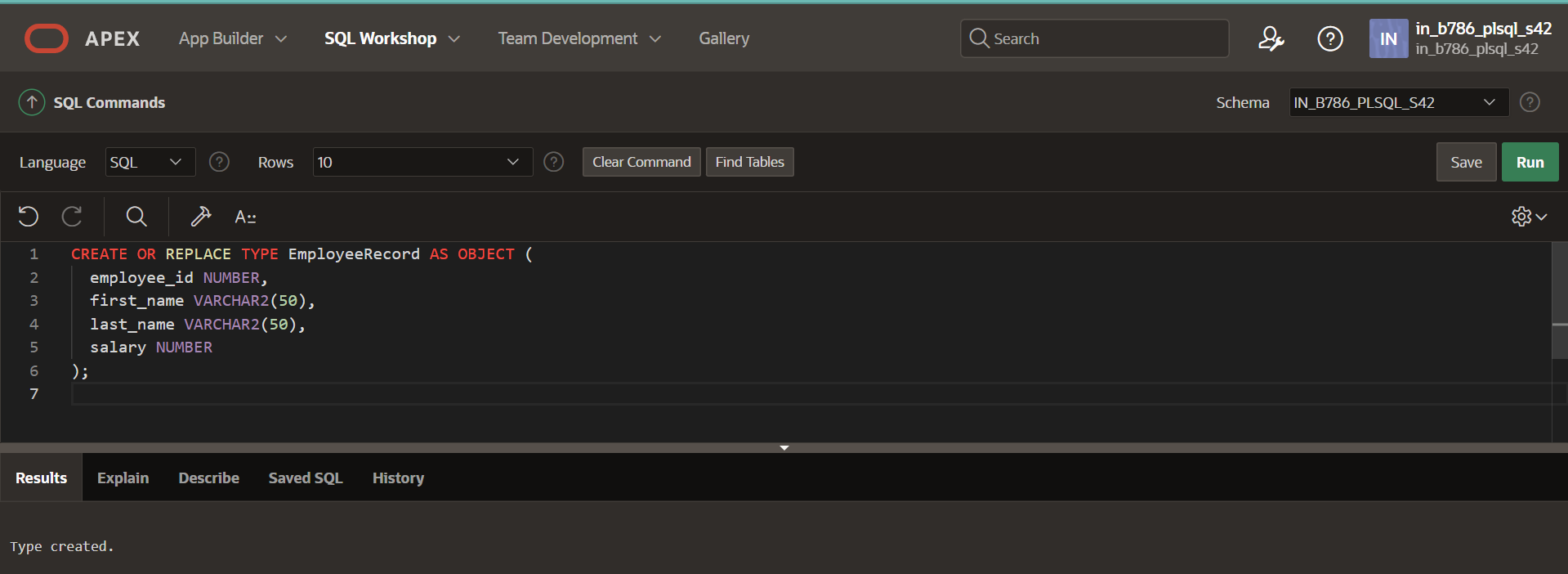
## EXCEPTION

## WHEN OTHERS THEN

## DBMS\_OUTPUT.PUT\_LINE('Error: ' || SQLERRM);

## END;

**Output**

****

## 

## 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 ref\_cursor\_type IS REF CURSOR;

## employee\_ref\_cursor ref\_cursor\_type;

## v\_sql VARCHAR2(1000);

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

## v\_employee\_id Employees.Emp\_ID%TYPE;

## v\_emp\_name Employees.emp\_name%TYPE;

## BEGIN

## -- Construct the dynamic SQL statement

## v\_sql := 'SELECT Emp\_ID, emp\_name FROM Employees WHERE Salary > :salary\_threshold';

## 

## -- Open the cursor variable with the dynamic SQL statement

## OPEN employee\_ref\_cursor FOR v\_sql USING v\_salary\_threshold;

## 

## -- Fetch the rows from the cursor

## LOOP

## FETCH employee\_ref\_cursor INTO v\_employee\_id, v\_emp\_name;

## EXIT WHEN employee\_ref\_cursor%NOTFOUND;

## -- Process each row (here we just display it)

## DBMS\_OUTPUT.PUT\_LINE('EmployeeID: ' || v\_employee\_id || ', Name: ' || v\_emp\_name);

## END LOOP;

## -- Close the cursor

## CLOSE employee\_ref\_cursor;

## EXCEPTION

## WHEN OTHERS THEN

## IF employee\_ref\_cursor%ISOPEN THEN

## CLOSE employee\_ref\_cursor;

## END IF;

## RAISE;

## 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:

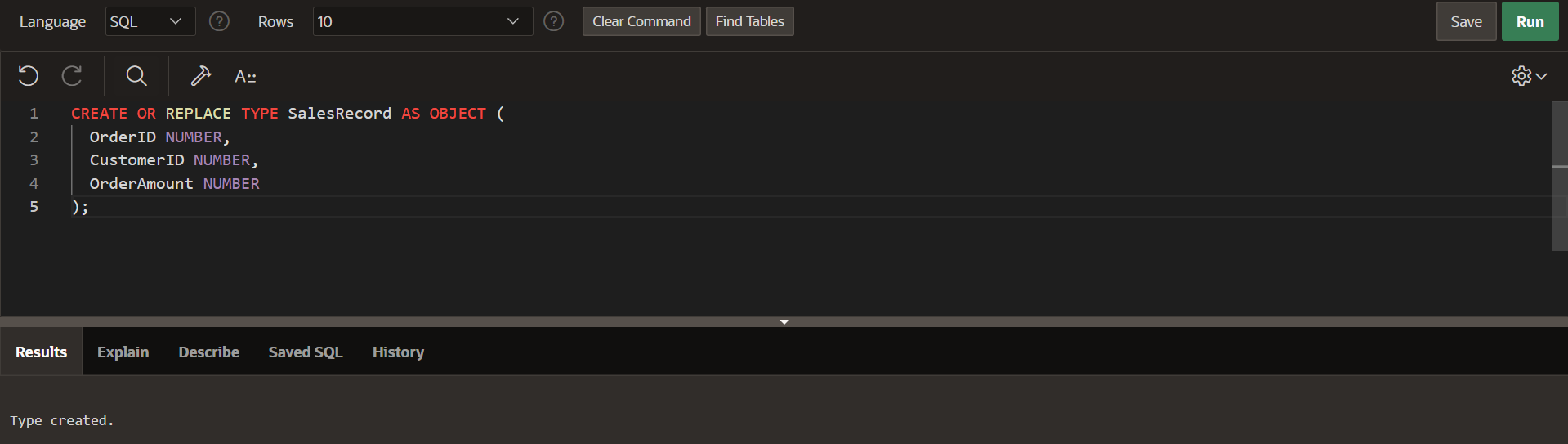
CREATE OR REPLACE TYPE SalesRecord AS OBJECT (

OrderID NUMBER,

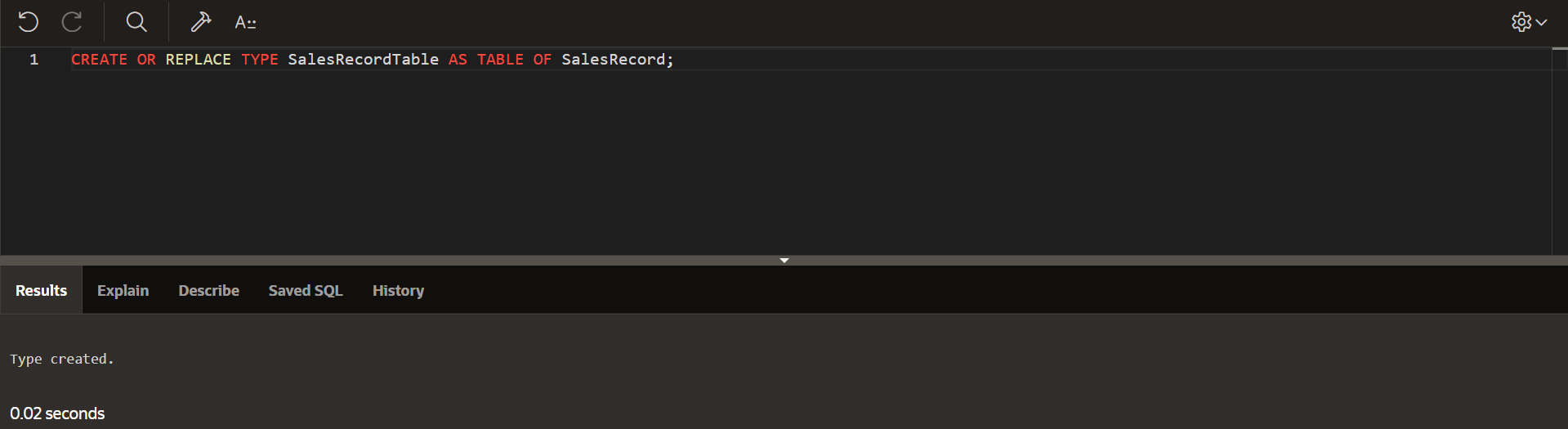
CustomerID NUMBER,

OrderAmount NUMBER

);



CREATE OR REPLACE TYPE SalesRecordTable AS TABLE OF SalesRecord;



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

RETURN SalesRecordTable PIPELINED

IS

CURSOR sales\_cur IS

SELECT OrderID, CustomerID, OrderAmount

FROM Orders

WHERE EXTRACT(MONTH FROM OrderDate) = p\_month

AND EXTRACT(YEAR FROM OrderDate) = p\_year;

sales\_rec SalesRecord;

BEGIN

OPEN sales\_cur;

LOOP

FETCH sales\_cur INTO sales\_rec.OrderID, sales\_rec.CustomerID, sales\_rec.OrderAmount;

EXIT WHEN sales\_cur%NOTFOUND;

PIPE ROW(sales\_rec);

END LOOP;

CLOSE sales\_cur;

RETURN;

EXCEPTION

WHEN OTHERS THEN

IF sales\_cur%ISOPEN THEN

CLOSE sales\_cur;

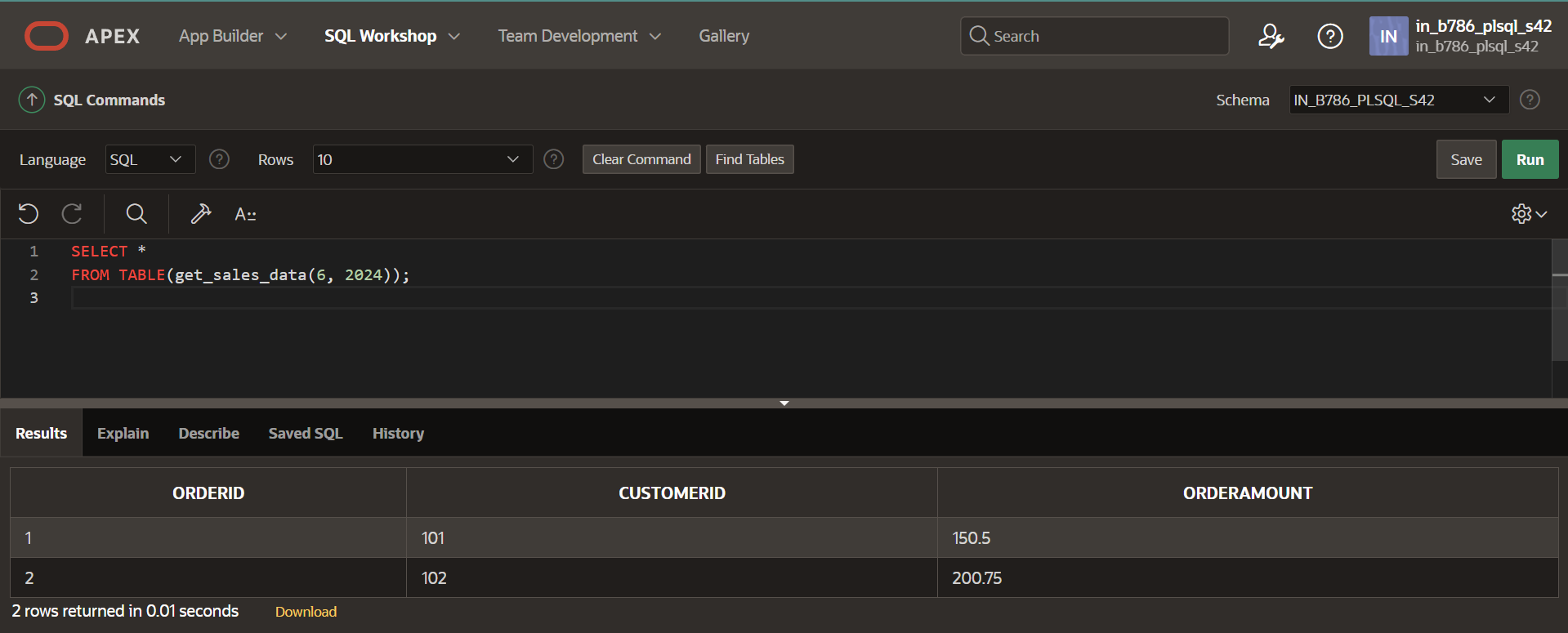
END IF;

RAISE;

END;

SELECT \*

FROM TABLE(get\_sales\_data(6, 2024));



## 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.