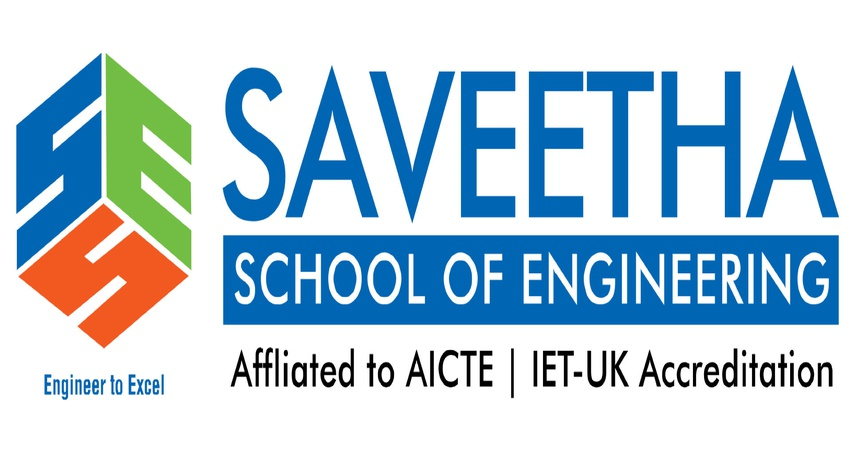
****

**Assignment - 3**

**SAVEETHA SCHOOL OF ENGINEERING**



Submitted by

**PADMANABHAN S (192324076)**

Submitted to

**Dr. Christy Melwyn**

Professor

Course Code: **CSA0556**

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

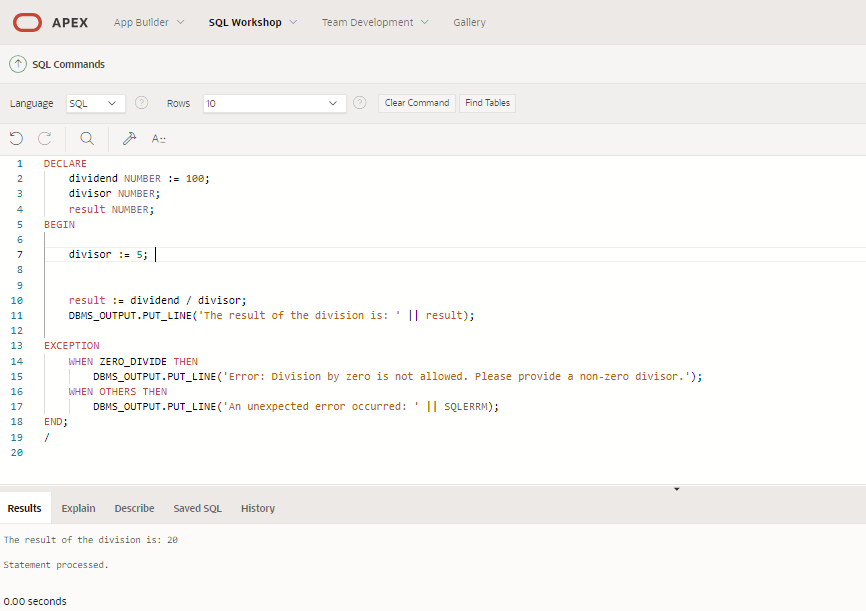
**Question 1: Handling Division Operation**

**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.**

QUERY:

IMPLEMENTATION:

****

DECLARE

dividend NUMBER := 100; -- You can set the dividend to any value

divisor NUMBER;

result NUMBER;

BEGIN

-- Prompt the user to enter the divisor

DBMS\_OUTPUT.PUT\_LINE('Enter the divisor: ');

-- Assuming the input is taken using some front-end application or procedure

-- Replace &divisor with actual user input handling in your application

divisor := &divisor;

-- Perform the division operation

result := dividend / divisor;

DBMS\_OUTPUT.PUT\_LINE('The result of the division is: ' || result);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

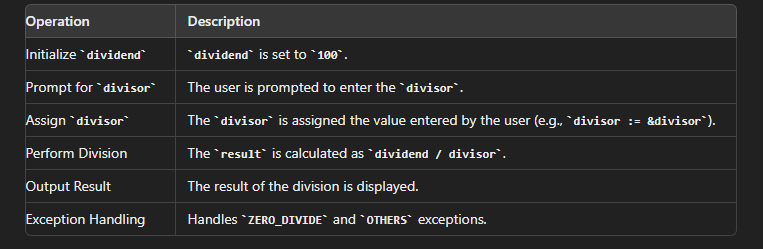
DBMS\_OUTPUT.PUT\_LINE('Error: Division by zero is not allowed. Please provide a non-zero divisor.');

WHEN OTHERS THEN

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

END;

/

****

****

**Query Explanation:**

DECLARE section:

* Declares two variables: 'dividend' and 'divisor' as NUMBER data types.
* Initializes 'dividend' with the value 100.

BEGIN section:

* Sets the value of 'divisor' to 5.

Calculation:

* Performs division: 'result := dividend / divisor;'
* This calculates 100 / 5, which equals 20.

Output:

* Uses DBMS\_OUTPUT.PUT\_LINE to display the result of the division.

Exception handling:

* WHEN ZERO\_DIVIDE THEN: Catches division by zero errors. If 'divisor' is 0, it will display an error message.
* WHEN OTHERS THEN: Catches any other unexpected errors. It will display a generic error message along with the specific SQL error (SQLERRM).

END: Closes the PL/SQL block.

1. It declares two number variables: 'dividend' set to 100, and 'divisor' which is later set to 5.
2. It performs the division of dividend by divisor (100 / 5).
3. It outputs the result of this division using DBMS\_OUTPUT.PUT\_LINE.
4. The query includes error handling:
   * It checks for division by zero and provides a specific error message if that occurs.
   * It also has a general error handler for any other unexpected errors.
5. The result shown at the bottom of the image (20) indicates that the query executed successfully, dividing 100 by 5.

This query demonstrates basic PL/SQL structure including variable declaration, arithmetic operations, output commands, and exception handling.

**Explanation of Error Handling:**

1. **ZERO\_DIVIDE Exception Handling**:
   * The ZERO\_DIVIDE exception is explicitly handled using the WHEN ZERO\_DIVIDE THEN clause. This exception occurs when the divisor is zero. The program catches this exception and outputs a user-friendly error message, informing the user that division by zero is not allowed.
2. **OTHERS Exception Handling**:
   * The WHEN OTHERS THEN clause is a catch-all for any other exceptions that might occur. This ensures that the program can handle unexpected errors gracefully. The SQLERRM function is used to display the error message associated with the exception.
3. **User Input Handling**:
   * In a real application, user input would typically be handled through a front-end interface. In this example, &divisor is used as a placeholder to represent user input. Ensure that your application captures and validates this input appropriately.

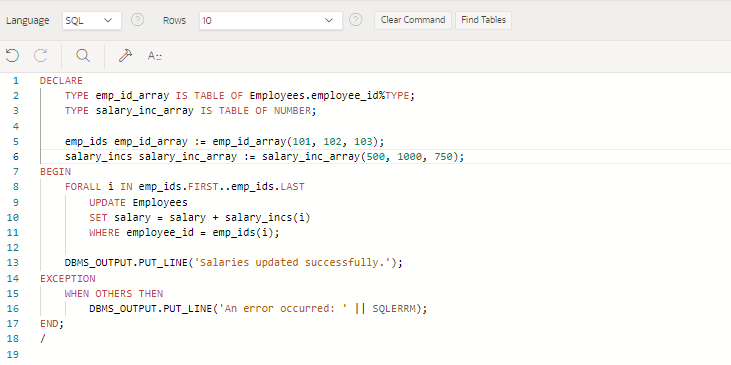
This PL/SQL block demonstrates how to handle exceptions in a structured manner, providing clear feedback to the user and ensuring the program can continue running or terminate gracefully in case of errors.

**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.**

**QUERY:**

****

**DECLARE**

**TYPE emp\_id\_array IS TABLE OF Employees.employee\_id%TYPE;**

**TYPE salary\_inc\_array IS TABLE OF NUMBER;**

**emp\_ids emp\_id\_array := emp\_id\_array(101, 102, 103); -- Example employee IDs**

**salary\_incs salary\_inc\_array := salary\_inc\_array(500, 1000, 750); -- Corresponding salary increments**

**BEGIN**

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

**UPDATE Employees**

**SET salary = salary + salary\_incs(i)**

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

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

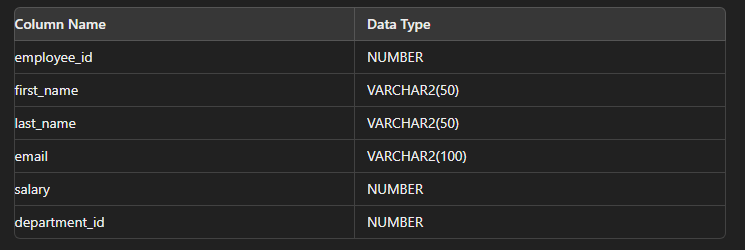
**EXCEPTION**

**WHEN OTHERS THEN**

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

**END;**

**/**

****

**Explanation:**

It declares two custom types:

* emp\_id\_array: A table type based on the Employees.employee\_id column type
* salary\_inc\_array: A table type of NUMBER values

It initializes two nested tables:

* emp\_ids: Contains employee IDs (101, 102, 103)
* salary\_incs: Contains salary increments (500, 1000, 750)

The main logic is in a FORALL loop:

* It iterates over the emp\_ids array
* For each employee ID, it updates the corresponding employee's salary in the Employees table
* The salary is increased by the amount specified in the salary\_incs array at the same index

After the update, it prints "Salaries updated successfully." using DBMS\_OUTPUT.PUT\_LINE

There's an exception handler for any unexpected errors, which would print the error message

**Bulk Processing with FORALL:**

* + **T**he FORALL statement in PL/SQL is used for bulk DML operations (INSERT, UPDATE, DELETE) on collections.
  + It allows you to execute a DML statement for each element in a collection with a single context switch between the PL/SQL and SQL engines.
  + This is particularly useful when dealing with large data sets, as it reduces the number of context switches and thereby improves performance.

1. **Improved Performance:**
   * Reduced Context Switching: In a typical loop, each DML statement execution requires a context switch between the PL/SQL and SQL engines. This can be costly in terms of performance, especially when dealing with a large number of records. FORALL minimizes these context switches by executing the DML operation in bulk.
   * Efficient Array Processing: FORALL processes entire arrays in one go, rather than row-by-row, which is more efficient. This is particularly beneficial when the operation involves a large number of rows, as it can significantly reduce the execution time.
   * Optimized Network Usage: When the PL/SQL engine and the SQL engine are on different systems, FORALL can reduce network traffic by sending fewer messages between the client and the database server.

Overall, using FORALL for bulk updates can lead to substantial performance improvements in applications that need to perform large-scale data modifications.

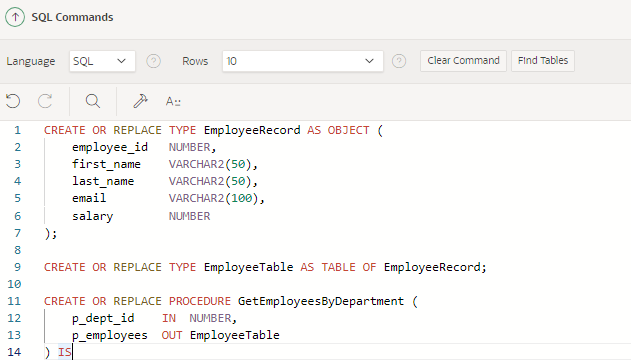
**Question 3: Implementing Nested Table Procedure**

**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.**

QUERY:

Table Creation:



CREATE OR REPLACE TYPE EmployeeRecord AS OBJECT (

employee\_id NUMBER,

first\_name VARCHAR2(50),

last\_name VARCHAR2(50),

email VARCHAR2(100),

salary NUMBER

);

CREATE OR REPLACE TYPE EmployeeTable AS TABLE OF EmployeeRecord;

CREATE OR REPLACE PROCEDURE GetEmployeesByDepartment (

p\_dept\_id IN NUMBER,

p\_employees OUT EmployeeTable

) IS

BEGIN

p\_employees := EmployeeTable();

-- Fetch employee details for the given department

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

BULK COLLECT INTO p\_employees

FROM Employees

WHERE department\_id = p\_dept\_id;

IF p\_employees.COUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('No employees found in department ' || p\_dept\_id);

END IF;

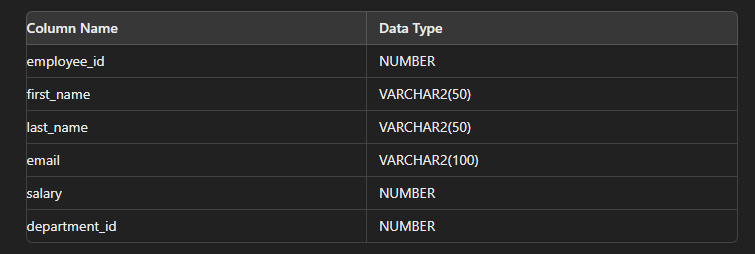
EXCEPTION

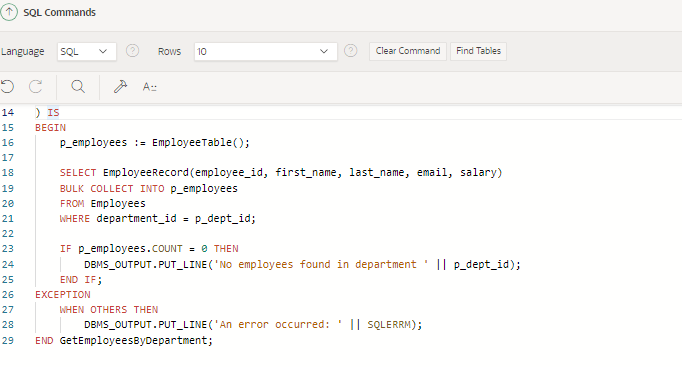
WHEN OTHERS THEN

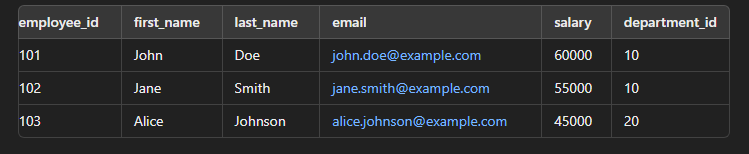
DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

END GetEmployeesByDepartment;

/



Implementation:  




**Explanation:**

By declaring a variable p\_employees of type EmployeeTable (which is likely a custom-defined table type to hold employee records).

It then uses a BULK COLLECT operation to retrieve employee data:

* The SELECT statement gets employee\_id, first\_name, last\_name, email, and salary.
* It collects this data into the p\_employees variable.
* The data is selected from the Employees table.
* It filters employees based on a specific department\_id (p\_dept\_id, which is likely a parameter passed to the procedure).

After the BULK COLLECT, it checks if any employees were found:

* IF p\_employees.COUNT = 0 THEN: This checks if the collection is empty.
* If no employees are found, it outputs a message saying "No employees found in department" along with the department ID.

The code includes exception handling:

* WHEN OTHERS THEN: This catches any unexpected errors.
* If an error occurs, it outputs "An error occurred:" followed by the specific error message (SQLERRM).

The **END statement closes the GetEmployeesByDepartment procedure.**

1. **Nested Table Type Definition**:
   * The EmployeeRecord object type is defined to store individual employee details, including employee\_id, first\_name, last\_name, email, and salary.
   * The EmployeeTable type is a nested table that stores a collection of EmployeeRecord objects. This nested table type allows for storing and handling multiple employee records as a single collection.
2. **Procedure GetEmployeesByDepartment**:
   * **Input Parameter (p\_dept\_id)**: This parameter accepts the department ID to filter the employees.
   * **Output Parameter (p\_employees)**: This parameter is used to return the collection of employees belonging to the specified department.
   * **Fetching Employees**: The procedure uses a SELECT statement with BULK COLLECT INTO to fetch all employee records that match the given department ID. The results are stored in the p\_employees nested table.
   * **Handling No Employees Found**: If no employees are found for the specified department, a message is printed using DBMS\_OUTPUT.PUT\_LINE.

### **Utilization and Returning of Nested Tables**

1. **Nested Tables as Output**:
   * Nested tables are a PL/SQL collection type that can be used to store multiple records or elements. They are similar to arrays but can be sparse and allow for easy manipulation, such as adding or deleting elements.
   * In this procedure, the nested table p\_employees is populated with employee records and returned as an output parameter. This allows the calling environment to receive and process a collection of employee data in a structured format.
2. **Advantages of Nested Tables**:
   * Nested tables provide a flexible way to handle sets of data in PL/SQL. They can be used in bulk operations, such as BULK COLLECT, and can be passed between procedures and functions.
   * They are also useful for returning multiple rows of data from a procedure, as demonstrated in this example, where all employees from a specific department are returned as a single collection.

This procedure and the use of nested tables provide an efficient way to handle and return sets of related data in PL/SQL applications.

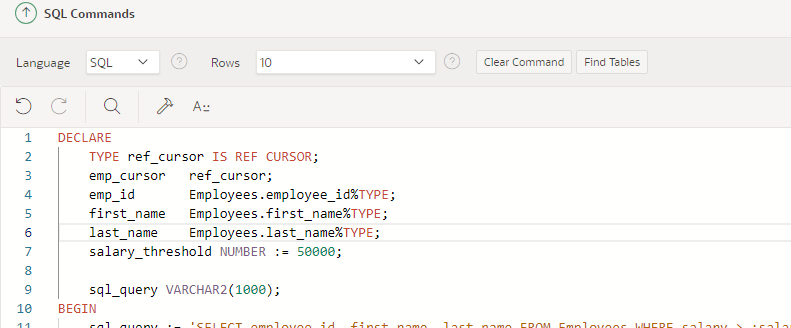
**Question 4: Using Cursor Variables and Dynamic SQL**

**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.**

QUERY:

TABLE CREATION:



**DECLARE**

**TYPE ref\_cursor IS REF CURSOR;**

**emp\_cursor ref\_cursor;**

**emp\_id Employees.employee\_id%TYPE;**

**first\_name Employees.first\_name%TYPE;**

**last\_name Employees.last\_name%TYPE;**

**salary\_threshold NUMBER := 50000; -- Set the desired salary threshold**

**sql\_query VARCHAR2(1000);**

**BEGIN**

**-- Construct the dynamic SQL query**

**sql\_query := 'SELECT employee\_id, first\_name, last\_name FROM Employees WHERE salary > :salary';**

**-- Open the cursor using dynamic SQL**

**OPEN emp\_cursor FOR sql\_query USING salary\_threshold;**

**-- Fetch and display the results**

**LOOP**

**FETCH emp\_cursor INTO emp\_id, first\_name, last\_name;**

**EXIT WHEN emp\_cursor%NOTFOUND;**

**DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || emp\_id || ', Name: ' || first\_name || ' ' || last\_name);**

**END LOOP;**

**-- Close the cursor**

**CLOSE emp\_cursor;**

**EXCEPTION**

**WHEN OTHERS THEN**

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

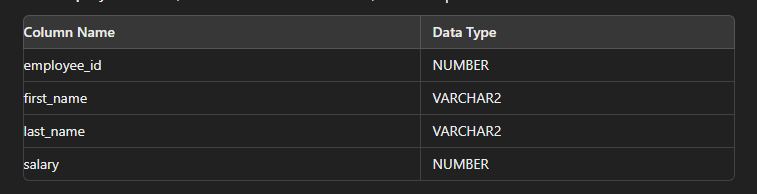
**IF emp\_cursor%ISOPEN THEN**

**CLOSE emp\_cursor;**

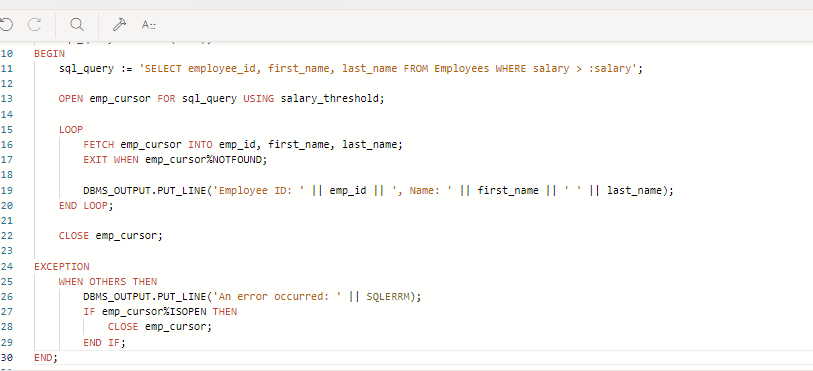
**END IF;**

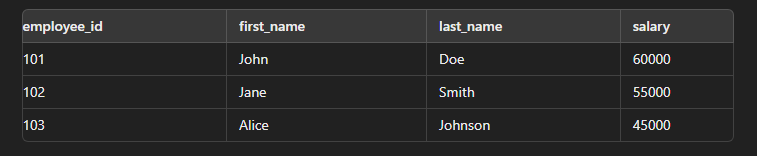
**END;**

**/**



IMPLEMENTATION:

****

**  
EXPLANATION:**

I start by defining a SQL query that selects employee\_id, first\_name, and last\_name from the Employees table where the salary is greater than a certain threshold (>:salary).

I open a cursor named emp\_cursor for this query, using a salary\_threshold parameter.

I then enter a loop where I:

* Fetch each row from the cursor into variables emp\_id, first\_name, and last\_name
* Exit the loop when there are no more rows (emp\_cursorNotFound)
* For each fetched row, I output the employee's ID and full name using DBMS\_OUTPUT.PUT\_LINE

After processing all rows, I close the cursor.

I have an exception handling block where I:

1. Catch any unexpected errors and output an error message
2. Ensure the cursor is closed if it's still open when an exception occurs
3. **Dynamic SQL**:
   * Dynamic SQL refers to SQL statements that are constructed and executed at runtime rather than being hard-coded. This allows for more flexibility, such as executing different queries based on user input or other runtime conditions.
   * In this example, the dynamic SQL query is constructed as a string in the sql\_query variable, which selects employee\_id, first\_name, and last\_name from the Employees table where the salary is above a certain threshold.
4. **Cursor Variables (REF CURSOR)**:
   * A cursor variable, defined using the REF CURSOR type, is a pointer to a result set that can be dynamically opened and closed. It allows for more flexible query execution, especially when the exact query is not known until runtime.
   * In this block, emp\_cursor is declared as a cursor variable of type ref\_cursor. The cursor is opened using the dynamic SQL query stored in sql\_query with the salary\_threshold passed as a bind variable using the USING clause.
5. **Using the Cursor**:
   * Once the cursor is open, the program fetches the results into local variables (emp\_id, first\_name, last\_name) using a loop. The FETCH statement retrieves one row at a time, and EXIT WHEN emp\_cursor%NOTFOUND exits the loop when there are no more rows to fetch.
   * After processing the result set, the cursor is closed using the CLOSE statement to release the associated resources.
6. **Handling Exceptions**:
   * The EXCEPTION block ensures that any errors during the execution are caught and handled appropriately. If an error occurs and the cursor is still open, it is closed within the exception handler to prevent resource leakage.

Dynamic SQL, combined with cursor variables, provides powerful capabilities in PL/SQL, enabling flexible and dynamic querying of data. This is particularly useful in scenarios where the exact query structure depends on user input or other runtime conditions.

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

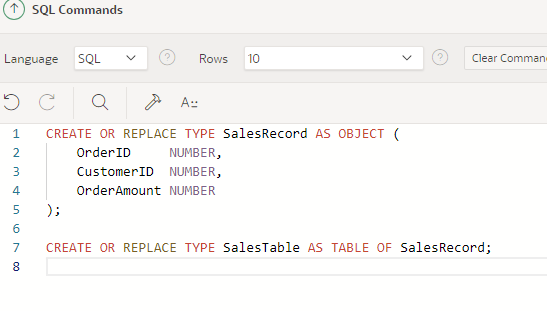
**Task:**

**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.**

QUERY:

TABLE CREATION:

### **1. Define the Record Type and Table Type:**



PL/SQL function get\_sales\_data that retrieves sales data for a given month and year. The function will return a table of records containing OrderID, CustomerID, and OrderAmount for orders placed in the specified month and year:

**CREATE OR REPLACE TYPE SalesRecord AS OBJECT (**

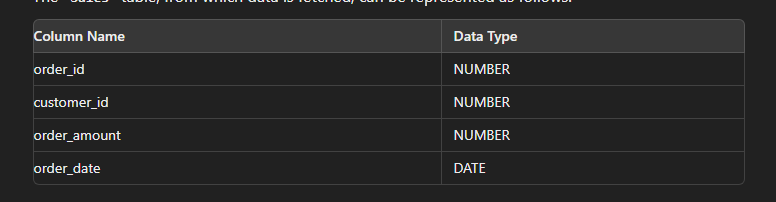
**OrderID NUMBER,**

**CustomerID NUMBER,**

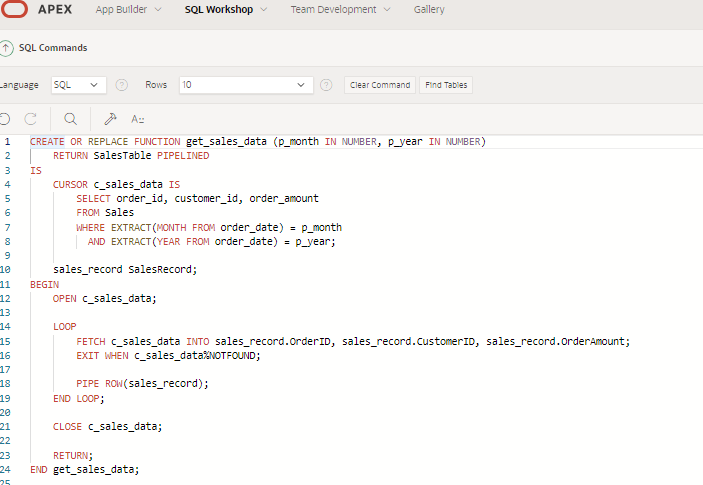
**OrderAmount NUMBER**

**);**

**CREATE OR REPLACE TYPE SalesTable AS TABLE OF SalesRecord;**

****

**2. Creating the Pipelined Function:**



**-- Pipelined function to retrieve sales data**

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

**RETURN SalesTable PIPELINED**

**IS**

**CURSOR c\_sales\_data IS**

**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;**

**sales\_record SalesRecord;**

**BEGIN**

**OPEN c\_sales\_data;**

**LOOP**

**FETCH c\_sales\_data INTO sales\_record.OrderID, sales\_record.CustomerID, sales\_record.OrderAmount;**

**EXIT WHEN c\_sales\_data%NOTFOUND;**

**PIPE ROW(sales\_record);**

**END LOOP;**

**CLOSE c\_sales\_data;**

**RETURN;**

**END get\_sales\_data;**

**/**

#### 

#### 

#### **Example Query Using the Pipelined Function:**

SELECT \*

FROM TABLE(get\_sales\_data(7, 2023));

This query retrieves sales data for July 2023 by calling the get\_sales\_data function and treating its output as a table.

**EXPLANATION:**

### **Explanation of Pipelined Table Functions**

#### **Pipelined Table Functions**

* **Definition**: Pipelined table functions return data as rows are being processed, rather than returning the entire result set at once. This allows the calling query to start processing rows before the entire function completes, improving overall efficiency and reducing memory usage.
* **Usage**: Pipelined functions are defined using the PIPELINED keyword and return data using the PIPE ROW statement within a loop. The function behaves like a table and can be queried using SQL.

#### **Benefits of Pipelined Table Functions**

1. **Efficiency in Data Retrieval**:
   * **Streaming Data**: By returning rows as they are processed, pipelined functions can start sending data to the client or calling query immediately. This is particularly useful for large data sets, where waiting for the entire result set to be processed before returning any data can be inefficient.
   * **Reduced Memory Consumption**: Since the function does not need to hold the entire result set in memory before returning it, memory usage is minimized. This is especially beneficial for functions that process large volumes of data.
2. **Parallel Execution**:
   * **Parallel Processing**: Pipelined functions can be parallelized, allowing different parts of the function to execute simultaneously. This can lead to significant performance improvements for data-intensive operations.
3. **Flexible Query Integration**:
   * **Seamless Integration**: Pipelined functions can be used in SQL queries as if they were regular tables. This allows for flexible integration with other SQL operations and makes them versatile tools for complex data processing tasks.

By using pipelined table functions, you can improve the efficiency and performance of data retrieval operations, especially when dealing with large datasets or when you need to start processing data as soon as possible.