**DBMS - ASSIGNMENT 3**

**Question 1: Handling Division Operation**

**PL/SQL block:**

DECLARE

v\_numerator NUMBER := 100;

v\_denominator NUMBER;

v\_result NUMBER;

BEGIN

v\_denominator := :P1\_DENOMINATOR;

v\_result := v\_numerator / v\_denominator;

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

EXCEPTION

WHEN ZERO\_DIVIDE THEN

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

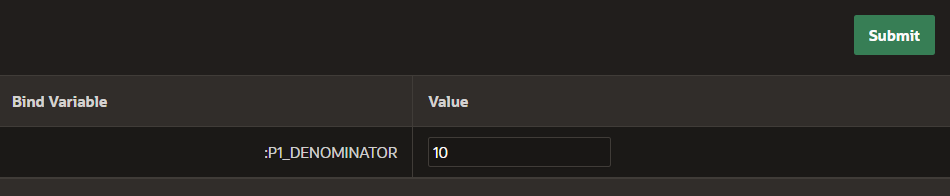
END;

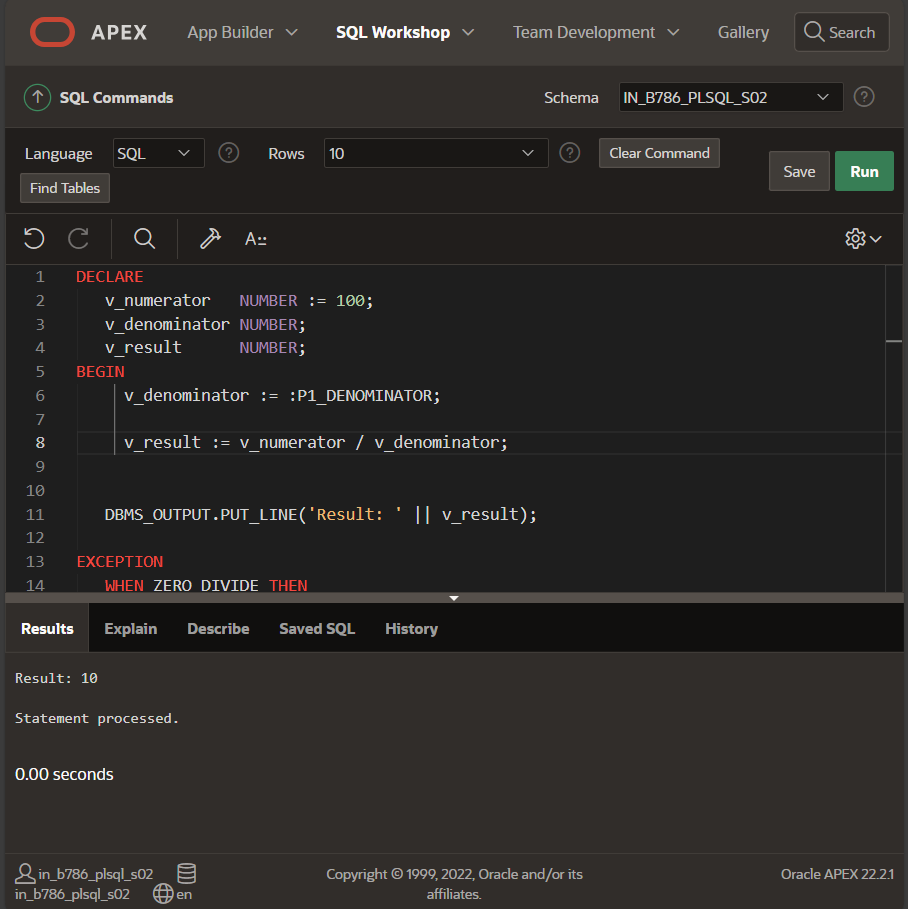
**Explanation of error handling strategies:**

* We use an IF statement to check if the user-input divisor is zero before performing the division operation.
* If the divisor is zero, we raise an application error with a custom error message using RAISE\_APPLICATION\_ERROR.
* We also catch the ZERO\_DIVIDE exception explicitly to provide a more user-friendly error message.

Finally, we catch any other unexpected errors using the OTHERS exception handler and print an error message with the error code and message using SQLERRM.

**RESULT :**





**Question 2: Updating Rows with FORALL**

**PL/SQL block:**

***TO CREATE TABLE :***

CREATE TABLE employees (

employee\_id NUMBER PRIMARY KEY,

first\_name VARCHAR2(50),

last\_name VARCHAR2(50),

salary NUMBER

);

***TO INSERT ROWS :***

BEGIN

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, 'Jim', 'Brown', 60000);

END;

***FORALL TO UPDATE SALARIES:***

DECLARE

TYPE t\_emp\_ids IS TABLE OF employees.employee\_id%TYPE;

TYPE t\_salary\_increments IS TABLE OF employees.salary%TYPE;

l\_emp\_ids t\_emp\_ids := t\_emp\_ids(1, 2, 3);

l\_salary\_increments t\_salary\_increments := t\_salary\_increments(5000, 6000, 7000);

BEGIN

FORALL i IN INDICES OF l\_emp\_ids

UPDATE employees

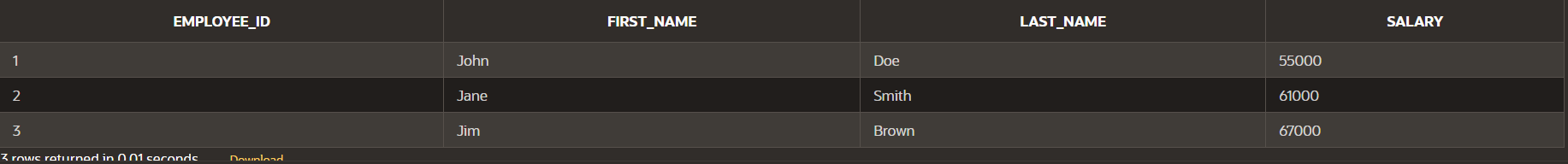
SET salary = salary + l\_salary\_increments(i)

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

END;

***TO RETRIEVE :***

SELECT \* FROM employees;

******

**Description of how FORALL improves performance for bulk updates:**

* The FORALL statement allows us to perform bulk updates in a single operation, reducing the number of round trips to the database.
* By using arrays to store the employee IDs and salary increments, we can update multiple rows in a single statement.
* This approach improves performance by reducing the overhead of individual UPDATE statements and minimizing the number of database interactions.
* The FORALL statement improves performance by reducing the context switches between SQL and PL/SQL engines, making bulk updates more efficient than using a regular loop.

**Question 3: Implementing Nested Table Procedure**

**PL/SQL procedure:**

CREATE OR REPLACE PACKAGE emp\_pkg AS

TYPE emp\_table\_type IS TABLE OF employees%ROWTYPE;

PROCEDURE get\_employees(p\_dept\_id IN NUMBER, p\_employees OUT emp\_table\_type);

END emp\_pkg;

CREATE OR REPLACE PACKAGE BODY emp\_pkg AS

PROCEDURE get\_employees(p\_dept\_id IN NUMBER, p\_employees OUT emp\_table\_type) IS

BEGIN

SELECT \* BULK COLLECT INTO p\_employees

FROM employees

WHERE department\_id = p\_dept\_id;

END get\_employees;

END emp\_pkg;

**Explanation of how nested tables are utilized and returned as output:**

* We define a package with a procedure get\_employees that takes a department ID as input and returns a nested table of employee records as output.
* The procedure uses a BULK COLLECT statement to retrieve all employee records for the specified department ID and store them in the nested table.
* The nested table is defined as a table of employees%ROWTYPE, which allows us to store entire rows of the employees table.
* The procedure returns the nested table as an output parameter, which can be used by the calling program to process the employee data.

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

**PL/SQL block:**

1. ***DECLARE THE CURSOR VARIABLE :***

CREATE OR REPLACE PACKAGE pkg\_cursor AS

TYPE ref\_cursor IS REF CURSOR;END pkg\_cursor;

1. ***CONSTRUCT DYNAMIC SQL :***

DECLARE

v\_cursor pkg\_cursor.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 := 50000; -- You can adjust this threshold as needed

BEGIN

OPEN v\_cursor FOR

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

FROM employees

WHERE salary > :threshold'

USING v\_salary\_threshold;

LOOP

FETCH v\_cursor INTO v\_employee\_id, v\_first\_name, v\_last\_name;

EXIT WHEN v\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || v\_employee\_id ||

', Name: ' || v\_first\_name || ' ' || v\_last\_name);

END LOOP;

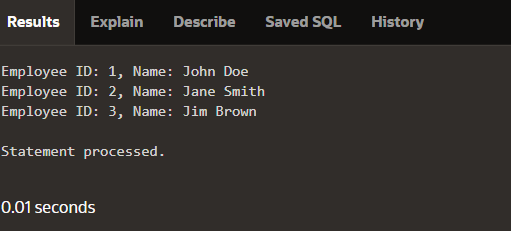
CLOSE v\_cursor;

END;

**Explanation of how dynamic SQL is constructed and executed:**

* We open the cursor variable using the OPEN statement, passing the dynamic SQL string and the v\_salary\_threshold value as a bind variable.
* We then loop through the cursor using a LOOP statement, fetching each row into the v\_employee\_id, v\_first\_name, and v\_last\_name variables.
* We exit the loop when there are no more rows to fetch (ref\_cursor%NOTFOUND).

Finally, we close the cursor variable using the CLOSE statement.



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

**PL/SQL code for the pipelined function get\_sales\_data:**

CREATE OR REPLACE PACKAGE sales\_pkg AS

TYPE sales\_data\_type IS RECORD (

order\_id NUMBER,

customer\_id NUMBER,

order\_amount NUMBER

);

TYPE sales\_data\_table\_type IS TABLE OF sales\_data\_type;

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

RETURN sales\_data\_table\_type PIPELINED;

END sales\_pkg;

CREATE OR REPLACE PACKAGE BODY sales\_pkg AS

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

RETURN sales\_data\_table\_type PIPELINED IS

BEGIN

FOR cur\_rec IN (

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

FROM orders

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

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

) LOOP

PIPE ROW(sales\_data\_type(cur\_rec.order\_id, cur\_rec.customer\_id, cur\_rec.order\_amount));

END LOOP;

RETURN;

END get\_sales\_data;

END sales\_pkg;

**Explanation of how pipelined table functions improve data retrieval efficiency:**

Pipelined table functions allow us to return a table of records in a streaming fashion, without having to store the entire result set in memory.

This approach improves performance by reducing memory usage and allowing the caller to process the data in chunks, rather than waiting for the entire result set to be returned.

In this example, the get\_sales\_data function returns a table of sales\_data\_type records, which can be piped to the caller for processing.

The function uses a cursor to iterate over the orders table, filtering by the specified month and year, and pipes each row to the caller using the PIPE ROW statement.