Question 1: Handling Division Operation

PL/SQL Block:

```plsql

DECLARE

num1 NUMBER := 100; -- Example numerator

num2 NUMBER; -- Divisor from user input

result NUMBER;

BEGIN

-- Simulating user input for the divisor

num2 := :divisor; -- Use a bind variable or get input in a real scenario

BEGIN

result := num1 / num2;

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

EXCEPTION

WHEN ZERO\_DIVIDE THEN

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

END;

END;

```

Explanation:

- The inner `BEGIN...END` block encapsulates the division operation.

- The `ZERO\_DIVIDE` exception is caught if the divisor is zero, and an appropriate error message is displayed using `DBMS\_OUTPUT.PUT\_LINE`.

- This approach ensures that the rest of the PL/SQL block can handle other logic without being interrupted by the exception.

Question 2: Updating Rows with FORALL

PL/SQL Block:

```plsql

DECLARE

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

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

l\_emp\_ids t\_emp\_ids := t\_emp\_ids();

l\_increments t\_increments := t\_increments();

BEGIN

-- Populate the arrays with employee IDs and corresponding salary increments

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

l\_increments := t\_increments(500, 1000, 1500); -- Example increments

FORALL i IN INDICES OF l\_emp\_ids

UPDATE employees

SET salary = salary + l\_increments(i)

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

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

END;

```

Explanation:

- `FORALL` improves performance by minimizing context switches between the PL/SQL and SQL engines.

- The `INDICES OF` clause ensures that all array elements are iterated over and the corresponding updates are performed in bulk, reducing the overhead of multiple individual updates.

Question 3: Implementing Nested Table Procedure

PL/SQL Procedure:

```plsql

CREATE OR REPLACE TYPE t\_emp\_record IS OBJECT (

employee\_id employees.employee\_id%TYPE,

first\_name employees.first\_name%TYPE,

last\_name employees.last\_name%TYPE

);

CREATE OR REPLACE TYPE t\_emp\_table IS TABLE OF t\_emp\_record;

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

p\_dept\_id IN departments.department\_id%TYPE,

p\_emp\_list OUT t\_emp\_table

) IS

BEGIN

SELECT t\_emp\_record(employee\_id, first\_name, last\_name)

BULK COLLECT INTO p\_emp\_list

FROM employees

WHERE department\_id = p\_dept\_id;

END;

```

Explanation:

- The `t\_emp\_record` object type defines the structure for employee records.

- The `t\_emp\_table` nested table type is a collection of `t\_emp\_record` objects.

- The procedure `get\_employees\_by\_dept` uses `BULK COLLECT` to fetch employees into a nested table and returns this collection as an output parameter.

Question 4: Using Cursor Variables and Dynamic SQL

PL/SQL Block:

```plsql

DECLARE

TYPE ref\_cursor IS REF CURSOR;

c\_emp 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 threshold

BEGIN

OPEN c\_emp FOR 'SELECT employee\_id, first\_name, last\_name FROM employees WHERE salary > :1' USING v\_salary\_threshold;

LOOP

FETCH c\_emp INTO v\_emp\_id, v\_first\_name, v\_last\_name;

EXIT WHEN c\_emp%NOTFOUND;

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

END LOOP;

CLOSE c\_emp;

END;

```

Explanation:

- A cursor variable (`ref\_cursor`) allows for dynamic SQL execution.

- The `OPEN...FOR` statement dynamically constructs and executes the SQL query based on the salary threshold.

- The `FETCH` loop retrieves results into variables, and `DBMS\_OUTPUT.PUT\_LINE` outputs the results.

Question 5: Designing Pipelined Function for Sales Data

PL/SQL Function:

```plsql

CREATE OR REPLACE TYPE t\_sales\_record IS OBJECT (

order\_id NUMBER,

customer\_id NUMBER,

order\_amount NUMBER

);

CREATE OR REPLACE TYPE t\_sales\_table IS TABLE OF t\_sales\_record;

CREATE OR REPLACE FUNCTION get\_sales\_data (

p\_month IN NUMBER,

p\_year IN NUMBER

) RETURN t\_sales\_table 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 (t\_sales\_record(r.order\_id, r.customer\_id, r.order\_amount));

END LOOP;

RETURN;

END;

```

Explanation:

- `t\_sales\_record` and `t\_sales\_table` define the structure and collection type for sales records.

- The `get\_sales\_data` function is a pipelined table function that uses `PIPE ROW` to stream results back to the caller.

- Pipelined functions improve performance by returning rows incrementally rather than all at once, which is useful for large datasets.