Chapter 17

Dynamic SQL in PL/SQL

Mayko Freitas da Silva

Introduction to Dynamic SQL

Welcome to our exploration of Dynamic SQL in PL/SQL. In this chapter, we'll learn how to create flexible and powerful database operations using dynamic SQL statements. We'll be working with Oracle's HR demonstration schema, which contains real-world examples of employee and department data.

What is Dynamic SQL?

Dynamic SQL is a programming technique that enables you to construct SQL statements at runtime rather than at compile time. Think of it as building a SQL statement like putting together building blocks - you can arrange them differently based on your needs at the moment of execution.

Why Use Dynamic SQL?

- When SQL statements need to be constructed based on user input
- When table names or column names are not known until runtime
- When you need to create flexible database operations

Topics We'll Cover:

1. Basic Dynamic SQL Execution
2. Working with Bind Variables
3. Dynamic DML Operations
4. Cursor Operations
5. Error Handling
6. Best Practices and Security

Let's start with a simple example using the HR schema:

```
DECLARE
   v_sql       VARCHAR2(200);
   v_dept_name  VARCHAR2(30);
   v_dept_id    NUMBER := 10;
BEGIN
   v_sql := 'SELECT department_name
           FROM departments
           WHERE department_id = :1';

   EXECUTE IMMEDIATE v_sql
   INTO v_dept_name
   USING v_dept_id;

   DBMS_OUTPUT.PUT_LINE('Department Name: ' || v_dept_name);
END;
```

Basic Syntax Structure

The basic structure of dynamic SQL includes:

1. Declaring variables to hold SQL statements
2. Constructing the SQL statement
3. Executing the statement using EXECUTE IMMEDIATE
4. Handling the results

Here's another example showing salary analysis:

```
DECLARE
   v_sql          VARCHAR2(500);
   v_avg_salary    NUMBER;
   v_department_id NUMBER := 50;  -- Shipping department
BEGIN
   v_sql := 'SELECT AVG(salary)
           FROM employees
           WHERE department_id = :1';

   EXECUTE IMMEDIATE v_sql
   INTO v_avg_salary
   USING v_department_id;

   DBMS_OUTPUT.PUT_LINE('Average Salary: $' ||
                TO_CHAR(v_avg_salary, '99,999.00'));
END;
```

Key Concepts to Remember:

1. Dynamic SQL statements are stored as strings
2. EXECUTE IMMEDIATE is the primary method for execution
3. Bind variables (like :1) help prevent SQL injection
4. The HR schema provides real business scenarios for practice

Safety and Security

When working with dynamic SQL, always remember:

- Validate input parameters
- Use bind variables instead of string concatenation
- Handle exceptions appropriately

Let's look at a more complex example that demonstrates these principles:

```
DECLARE
  v_sql         VARCHAR2(500);
  v_column_name  VARCHAR2(30) := 'salary';
  v_table_name   VARCHAR2(30) := 'employees';
  v_result       NUMBER;
BEGIN
  -- Validate table name
  IF v_table_name NOT IN ('employees', 'departments',
                      'jobs', 'locations') THEN
    RAISE_APPLICATION_ERROR(-20001, 'Invalid table name');
  END IF;

  -- Construct and execute dynamic query
  v_sql := 'SELECT AVG(' || v_column_name || ')
       FROM ' || v_table_name;

  EXECUTE IMMEDIATE v_sql INTO v_result;

  DBMS_OUTPUT.PUT_LINE('Average ' || v_column_name || ': ' ||
                    TO_CHAR(v_result, '99,999.00'));
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
```

Understanding Bind Variables and Complex Dynamic SQL

What Are Bind Variables?

Bind variables are placeholders in SQL statements that allow you to:

- Pass values to your dynamic SQL statements
- Improve performance through statement reuse
- Protect against SQL injection attacks

Think of bind variables as empty boxes that you'll fill with values when the SQL actually runs. They're like variables in a mathematical equation where you can plug in different values without changing the formula itself.

Types of Bind Variables:

1. Positional bind variables (:1, :2, :3)
2. Named bind variables (:employee_id, :salary)
3. IN, OUT, and IN OUT parameters

Let's explore this with practical examples using the HR schema:

```
DECLARE
   v_sql         VARCHAR2(500);
   v_employee_id  NUMBER := 103;  -- Alexander Hunold
   v_salary       NUMBER;
   v_job_title    VARCHAR2(35);
BEGIN
   -- Using multiple bind variables
   v_sql := 'SELECT salary, job_title
           FROM employees e JOIN jobs j ON (e.job_id = j.job_id)
           WHERE employee_id = :1';

   EXECUTE IMMEDIATE v_sql
   INTO v_salary, v_job_title
   USING v_employee_id;

   DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id);
   DBMS_OUTPUT.PUT_LINE('Job Title: ' || v_job_title);
   DBMS_OUTPUT.PUT_LINE('Salary: $' || TO_CHAR(v_salary, '99,999.00'));
END;
```

Dynamic PL/SQL Blocks

Sometimes you need to execute an entire PL/SQL block dynamically. Here's how:

```
DECLARE
    v_dynamic_block VARCHAR2(1000);
    v_employee_id   NUMBER := 103;
    v_salary        NUMBER;
BEGIN
    -- Get the salary using dynamic SQL
    v_dynamic_block := 'BEGIN
        SELECT salary
        INTO :1
        FROM employees
        WHERE employee_id = :2;
    END;';

    EXECUTE IMMEDIATE v_dynamic_block
    USING OUT v_salary, IN v_employee_id;

    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id);
    DBMS_OUTPUT.PUT_LINE('Current Salary: $' ||
                        TO_CHAR(v_salary, '99,999.00'));
    DBMS_OUTPUT.PUT_LINE('After 10% increase: $' ||
                        TO_CHAR(v_salary * 1.1, '99,999.00'));
END;
```

Error Handling in Dynamic SQL

Error handling is crucial when working with dynamic SQL. Here's how to handle common scenarios:

```
DECLARE
   v_sql         VARCHAR2(500);
   v_department_id NUMBER := 999; -- Non-existent department
   v_result       NUMBER;
   -- Custom exception
   e_no_data_found EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_no_data_found, -01403);
BEGIN
   v_sql := 'SELECT COUNT(*)
           FROM employees
           WHERE department_id = :1';

   EXECUTE IMMEDIATE v_sql
   INTO v_result
   USING v_department_id;

   DBMS_OUTPUT.PUT_LINE('Employee count: ' || v_result);
EXCEPTION
   WHEN e_no_data_found THEN
      DBMS_OUTPUT.PUT_LINE('No employees found in department ' ||
                     v_department_id);
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
```

Dynamic DDL Operations

Dynamic SQL allows you to perform DDL (Data Definition Language) operations. Here's an example creating a backup table:

```
DECLARE
   v_sql VARCHAR2(1000);
   v_table_exists NUMBER;
BEGIN
   -- Check if backup table exists
   SELECT COUNT(*)
   INTO v_table_exists
   FROM user_tables
   WHERE table_name = 'EMP_BACKUP';

   -- Drop table if exists
   IF v_table_exists > 0 THEN
      EXECUTE IMMEDIATE 'DROP TABLE emp_backup';
   END IF;

   -- Create backup table
   v_sql := 'CREATE TABLE emp_backup AS
           SELECT * FROM employees
           WHERE department_id = 60'; -- IT department

   EXECUTE IMMEDIATE v_sql;

   -- Get count of backed up records
   EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM emp_backup'
   INTO v_table_exists;

   DBMS_OUTPUT.PUT_LINE('Backed up ' || v_table_exists ||
                  ' IT department employees');
END;
```

Best Practices for Dynamic SQL

1. Use Bind Variables Properly:

```
-- Good Practice
v_sql := 'SELECT last_name FROM employees WHERE employee_id = :1';
-- Bad Practice (Never do this)
-- v_sql := 'SELECT last_name FROM employees WHERE employee_id = ' ||
--         v_employee_id;
```

2. Validate Input Parameters:

```
DECLARE
  v_sql      VARCHAR2(500);
  v_column   VARCHAR2(30) := 'SALARY';
BEGIN
  -- Validate column name
  IF v_column NOT IN (SELECT column_name
                  FROM user_tab_columns
                  WHERE table_name = 'EMPLOYEES') THEN
    RAISE_APPLICATION_ERROR(-20001, 'Invalid column name');
  END IF;

  v_sql := 'SELECT AVG(' || v_column || ') FROM employees';
  -- Continue with execution...
END;
```

Practice Exercise

Create a dynamic SQL block that:

1. Accepts a department ID
2. Returns the following department statistics:
    - Number of employees
    - Average salary
    - Minimum salary
    - Maximum salary
3. Handles cases where the department doesn't exist
4. Uses proper bind variables and error handling

Dynamic Cursors and Advanced Operations

Understanding Dynamic Cursors

In PL/SQL, a cursor is a pointer to a private SQL area that stores a parsed SQL statement and other processing information. When working with dynamic SQL, we can create cursors that are defined at runtime. This is particularly useful when:

- The number of columns in the result set is unknown until runtime
- The column types might vary
- You need to process multiple rows based on dynamic conditions

REF CURSOR Basics

A REF CURSOR is a cursor variable that can point to any query result set. Here's how to use them with the HR schema:

```
DECLARE
  -- Define REF CURSOR type
  TYPE emp_refcursor_type IS REF CURSOR;
  v_emp_cursor   emp_refcursor_type;
  v_sql          VARCHAR2(500);
  v_department_id NUMBER := 60; -- IT Department
  -- Record variables to store the fetched data
  v_first_name   employees.first_name%TYPE;
  v_last_name    employees.last_name%TYPE;
  v_salary       employees.salary%TYPE;
BEGIN
  -- Construct dynamic query
  v_sql := 'SELECT first_name, last_name, salary
       FROM employees
       WHERE department_id = :1
       ORDER BY salary DESC';

  -- Open cursor with dynamic SQL
  OPEN v_emp_cursor FOR v_sql USING v_department_id;

  DBMS_OUTPUT.PUT_LINE('IT Department Employees:');
  DBMS_OUTPUT.PUT_LINE('------------------------');

  -- Fetch and process results
  LOOP
    FETCH v_emp_cursor
    INTO v_first_name, v_last_name, v_salary;
    EXIT WHEN v_emp_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE(RPAD(v_first_name || ' ' || v_last_name, 30)
               || ' $' || TO_CHAR(v_salary, '99,999.00'));
  END LOOP;

  -- Close cursor
  CLOSE v_emp_cursor;
EXCEPTION
  WHEN OTHERS THEN
    IF v_emp_cursor%ISOPEN THEN
      CLOSE v_emp_cursor;
    END IF;
    RAISE;
END;
```

Dynamic SQL with Record Types

Using record types can make your code more maintainable when dealing with multiple columns:

```
DECLARE
  -- Define record type based on employees table
  TYPE emp_record_type IS RECORD (
    emp_id    employees.employee_id%TYPE,
    name      VARCHAR2(50),
    job       jobs.job_title%TYPE,
    dept_name departments.department_name%TYPE
  );

  v_emp_record emp_record_type;
  v_sql       VARCHAR2(500);

BEGIN
  v_sql := 'SELECT e.employee_id,
            e.first_name || '' '' || e.last_name,
            j.job_title,
            d.department_name
         FROM employees e
         JOIN jobs j ON (e.job_id = j.job_id)
         JOIN departments d ON (e.department_id = d.department_id)
         WHERE e.employee_id = :1';

  EXECUTE IMMEDIATE v_sql
  INTO v_emp_record
  USING 103; -- Alexander Hunold

  DBMS_OUTPUT.PUT_LINE('Employee Details:');
  DBMS_OUTPUT.PUT_LINE('ID: ' || v_emp_record.emp_id);
  DBMS_OUTPUT.PUT_LINE('Name: ' || v_emp_record.name);
  DBMS_OUTPUT.PUT_LINE('Job: ' || v_emp_record.job);
  DBMS_OUTPUT.PUT_LINE('Department: ' || v_emp_record.dept_name);
END;
```

Dynamic SQL with Multiple Result Sets

Here's how to handle multiple result sets dynamically:

```
DECLARE
   TYPE dept_refcursor_type IS REF CURSOR;
   v_dept_cursor    dept_refcursor_type;
   v_sql            VARCHAR2(500);
   v_min_salary     NUMBER := 5000;

   -- Record type for department statistics
   TYPE dept_stats_record IS RECORD (
      dept_name   departments.department_name%TYPE,
      emp_count   NUMBER,
      avg_salary  NUMBER
   );
   v_dept_stats    dept_stats_record;
BEGIN
   v_sql := 'SELECT d.department_name,
                 COUNT(*),
                 AVG(salary)
            FROM employees e
            JOIN departments d ON (e.department_id = d.department_id)
            WHERE salary >= :1
            GROUP BY d.department_name
            ORDER BY AVG(salary) DESC';

   OPEN v_dept_cursor FOR v_sql USING v_min_salary;

   DBMS_OUTPUT.PUT_LINE('Department Statistics (Salary >= $' ||
                   TO_CHAR(v_min_salary, '99,999.00') || '):');
   DBMS_OUTPUT.PUT_LINE('----------------------------------------');

   LOOP
      FETCH v_dept_cursor INTO v_dept_stats;
      EXIT WHEN v_dept_cursor%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE(
         RPAD(v_dept_stats.dept_name, 20) ||
         LPAD(v_dept_stats.emp_count, 5) || ' employees, ' ||
         'Avg Salary: $' ||
         TO_CHAR(v_dept_stats.avg_salary, '99,999.00')
      );
   END LOOP;

   CLOSE v_dept_cursor;
END;
```

Advanced Error Handling Techniques

Here's a comprehensive error handling approach:

```
DECLARE
  v_sql VARCHAR2(500);
  -- Custom exceptions
  e_invalid_department EXCEPTION;
  e_salary_too_high    EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_invalid_department, -20001);
  PRAGMA EXCEPTION_INIT(e_salary_too_high, -20002);

  -- Variables for employee update
  v_emp_id      NUMBER := 103;
  v_new_salary NUMBER := 50000;
  v_count       NUMBER;
BEGIN
  -- Validate department (correct way)
  v_sql := 'SELECT COUNT(*)
            FROM employees
            WHERE employee_id = :1';

  EXECUTE IMMEDIATE v_sql
  INTO v_count
  USING v_emp_id;

  IF v_count = 0 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Employee does not exist');
  END IF;

  -- Validate salary
  IF v_new_salary > 50000 THEN
    RAISE_APPLICATION_ERROR(-20002, 'Salary exceeds maximum allowed');
  END IF;

  -- If validation passes, update salary
  v_sql := 'UPDATE employees
            SET salary = :1
            WHERE employee_id = :2';

  EXECUTE IMMEDIATE v_sql
  USING v_new_salary, v_emp_id;

  COMMIT;

EXCEPTION
  WHEN e_invalid_department THEN
    DBMS_OUTPUT.PUT_LINE('Error: Invalid employee ID');
    ROLLBACK;
  WHEN e_salary_too_high THEN
    DBMS_OUTPUT.
```

```
PUT_LINE('Error: Salary exceeds maximum allowed');
    ROLLBACK;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM);
    ROLLBACK;
END;
```

If the last code generates an ORA-04091 error, it means there is a trigger that is not handling the issue with mutating tables correctly. Compound Triggers and Mutating Tables were covered in Module 14. However, here are some tips to try resolving the issue before reviewing Module 14.

1. First, let's find what triggers exist on the EMPLOYEES table:

```
SELECT trigger_name,
       trigger_type,
       triggering_event,
       status
FROM user_triggers
WHERE table_name = 'EMPLOYEES'
ORDER BY trigger_name;
```

2. Once you identify the triggers, you can see their definitions:

```
SELECT trigger_name,
       trigger_body
FROM user_triggers
WHERE table_name = 'EMPLOYEES';
```

3. To temporarily disable a trigger for testing:

```
-- Disable a specific trigger
ALTER TRIGGER trigger_name DISABLE;

-- Or disable all triggers on EMPLOYEES table
ALTER TABLE employees DISABLE ALL TRIGGERS;
```

4. After testing, remember to re-enable the triggers:

```
-- Enable a specific trigger
ALTER TRIGGER trigger_name ENABLE;

-- Or enable all triggers on EMPLOYEES table
ALTER TABLE employees ENABLE ALL TRIGGERS;
```

The error ORA-04091 typically occurs because:

- There's likely a trigger on EMPLOYEES table that fires on UPDATE
- When your code updates the salary, the trigger fires
- Inside that trigger, there's probably a query against the EMPLOYEES table
- This creates a "mutating table" situation

Here's how to create a COMPOUND trigger to fix the mutating table error. We'll need to create both the trigger and modify the original code:

1. First, create the COMPOUND trigger:

```
CREATE OR REPLACE TRIGGER trg_validate_salary_update
FOR UPDATE OF salary ON employees
COMPOUND TRIGGER
    -- Global variables for the trigger
    TYPE t_salary_rec IS RECORD (
        emp_id    employees.employee_id%TYPE,
        old_sal   employees.salary%TYPE,
        new_sal   employees.salary%TYPE
    );

    TYPE t_salary_tbl IS TABLE OF t_salary_rec INDEX BY PLS_INTEGER;
    g_salaries t_salary_tbl;
    g_idx PLS_INTEGER := 0;

    -- Constants
    g_max_salary CONSTANT NUMBER := 50000;

    -- Before each row section
    BEFORE EACH ROW IS
    BEGIN
        g_idx := g_idx + 1;
        g_salaries(g_idx).emp_id  := :NEW.employee_id;
        g_salaries(g_idx).old_sal := :OLD.salary;
        g_salaries(g_idx).new_sal := :NEW.salary;

        -- Validate new salary
        IF :NEW.salary > g_max_salary THEN
            RAISE_APPLICATION_ERROR(-20002, 'Salary exceeds maximum allowed');
        END IF;
    END BEFORE EACH ROW;

    -- After statement section
    AFTER STATEMENT IS
    BEGIN
        -- Additional validations if needed
        NULL;
    END AFTER STATEMENT;
END trg_validate_salary_update;
/
```

Key changes and explanations:

1. The COMPOUND trigger:

    - Uses BEFORE EACH ROW for immediate validations
    - Stores salary changes in a collection for potential use
    - Performs the salary validation before the update takes effect
    - Can handle multiple rows if needed
    - Avoids mutating table issues by using :NEW and :OLD pseudorecords

Benefits of this approach:

- Prevents mutating table errors
- Centralizes salary validation logic in the trigger
- Handles both single and bulk updates
- Maintains data integrity
- Provides better separation of concerns

To test the solution:

```
-- Test with valid salary
BEGIN
   UPDATE employees
   SET salary = 45000
   WHERE employee_id = 103;

   COMMIT;
   DBMS_OUTPUT.PUT_LINE('Update successful');
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
      ROLLBACK;
END;
/

-- Test with invalid salary
BEGIN
   UPDATE employees
   SET salary = 55000
   WHERE employee_id = 103;

   COMMIT;
   DBMS_OUTPUT.PUT_LINE('Update successful');
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
      ROLLBACK;
END;
/
```

Summary of Best Practices

1. Always close cursors in exception handlers
2. Use parameter validation before execution
3. Implement proper error handling
4. Use bind variables for better security
5. Consider performance implications
6. Document complex dynamic SQL thoroughly

Dynamic SQL in PL/SQL