Chapter 03

SQL in PL/SQL

Mayko Freitas da Silva

Introduction

Welcome to an advanced exploration of SQL integration within PL/SQL. This chapter delves into the symbiotic relationship between SQL and PL/SQL, focusing on how to leverage SQL's data manipulation and querying capabilities within PL/SQL's procedural framework.

PL/SQL, Oracle's procedural extension to SQL, allows for seamless embedding of SQL statements within its block structure. This integration enables developers to combine the declarative power of SQL with the procedural logic of PL/SQL, resulting in robust and efficient database applications.

Key technical aspects you'll encounter include:

1. Dynamic SQL execution using the SELECT INTO statement
2. Manipulation of database objects through DML operations within PL/SQL blocks
3. Utilization of sequence objects for generating unique identifiers
4. Implementation of transaction control mechanisms for data consistency and integrity

Throughout this chapter, we'll use Oracle's HR schema as our playground, providing practical, real-world examples that demonstrate these concepts in action.

In this chapter, you will learn about:

- SQL Statements in PL/SQL
- Transaction Control in PL/SQL

Let's begin by examining how SQL statements can be seamlessly integrated into PL/SQL blocks, enhancing your ability to interact with the Oracle database at a more granular and controlled level.

Lab 3.1: SQL Statements in PL/SQL

After this lab, you will be able to:

- Initialize Variables with the SELECT INTO Statement
- Use DML Statements in a PL/SQL Block
- Use a Sequence in a PL/SQL Block

Initializing Variables with SELECT INTO

The SELECT INTO statement is a powerful tool for retrieving data from the database and assigning it to PL/SQL variables. It's particularly useful for single-row queries.

Example 1: Basic SELECT INTO

```
DECLARE
  v_emp_name employees.first_name%TYPE;
  v_emp_salary employees.salary%TYPE;
BEGIN
  SELECT first_name, salary
  INTO v_emp_name, v_emp_salary
  FROM employees
  WHERE employee_id = 103;

  DBMS_OUTPUT.PUT_LINE('Employee: ' || v_emp_name || ', Salary: $' || v_emp_salary);
END;
```

Example 2: Using Aggregate Functions

```
DECLARE
  v_dept_id departments.department_id%TYPE := 60;
  v_avg_salary NUMBER(8,2);
  v_emp_count NUMBER;
BEGIN
  SELECT AVG(salary), COUNT(*)
  INTO v_avg_salary, v_emp_count
  FROM employees
  WHERE department_id = v_dept_id;

  DBMS_OUTPUT.PUT_LINE('Department ' || v_dept_id || ' stats:');
  DBMS_OUTPUT.PUT_LINE('Average salary: $' || v_avg_salary);
  DBMS_OUTPUT.PUT_LINE('Number of employees: ' || v_emp_count);
END;
```

Example 3: Handling No Data Found

```
DECLARE
  v_job_title jobs.job_title%TYPE;
  v_min_salary jobs.min_salary%TYPE;
BEGIN
  BEGIN
    SELECT job_title, min_salary
    INTO v_job_title, v_min_salary
    FROM jobs
    WHERE job_id = 'IT_PROG';

    DBMS_OUTPUT.PUT_LINE('Job: ' || v_job_title || ', Min Salary: $' || v_min_salary);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No job found with that ID.');
  END;
END;
```

Using DML Statements in PL/SQL

DML statements (INSERT, UPDATE, DELETE) can be seamlessly integrated into PL/SQL blocks, allowing for complex data manipulation operations.

Example 1: INSERT Statement

```
DECLARE
  v_emp_id employees.employee_id%TYPE;
  v_emp_email employees.email%TYPE;
BEGIN
  -- Get the maximum employee ID and add 1
  SELECT MAX(employee_id) + 1
  INTO v_emp_id
  FROM employees;

  -- Generate an email based on the new ID
  v_emp_email := 'EMP' || v_emp_id || '@example.com';

  -- Insert the new employee
  INSERT INTO employees (employee_id, first_name, last_name, email, hire_date, job_id)
  VALUES (v_emp_id, 'John', 'Doe', v_emp_email, SYSDATE, 'IT_PROG');

  DBMS_OUTPUT.PUT_LINE('New employee inserted with ID: ' || v_emp_id);

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

Example 2: UPDATE Statement with Subquery

```
DECLARE
  v_dept_id departments.department_id%TYPE := 30;
  v_salary_increase NUMBER := 1.1; -- 10% increase
  v_affected_rows NUMBER;
BEGIN
  UPDATE employees e
  SET salary = salary * v_salary_increase
  WHERE department_id = v_dept_id
  AND salary < (SELECT AVG(salary) FROM employees WHERE department_id = e.department_id);

  v_affected_rows := SQL%ROWCOUNT;

  DBMS_OUTPUT.PUT_LINE('Employees updated: ' || v_affected_rows);

  COMMIT;
END;
```

Example 3: DELETE Statement with RETURNING Clause

```
DECLARE
  v_emp_id employees.employee_id%TYPE := 206;
  v_emp_name employees.first_name%TYPE;
  v_emp_salary employees.salary%TYPE;
BEGIN
  DELETE FROM employees
  WHERE employee_id = v_emp_id
  RETURNING first_name, salary INTO v_emp_name, v_emp_salary;

  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Deleted employee: ' || v_emp_name || ', Salary: $' || v_emp_salary);
  ELSE
    DBMS_OUTPUT.PUT_LINE('No employee found with ID: ' || v_emp_id);
  END IF;

  COMMIT;
END;
```

Using Sequences in PL/SQL

Sequences provide a simple way to generate unique numeric identifiers, which is particularly useful for primary key columns.

Example 1: Creating and Using a Sequence

```
-- First, create the sequence
CREATE SEQUENCE emp_id_seq
START WITH 300
INCREMENT BY 1
NOCACHE
NOCYCLE;

-- Now, use it in a PL/SQL block
DECLARE
  v_emp_id employees.employee_id%TYPE;
  v_emp_email employees.email%TYPE;
BEGIN
  -- Get the next sequence value
  v_emp_id := emp_id_seq.NEXTVAL;
  v_emp_email := 'EMP' || v_emp_id || '@example.com';

  INSERT INTO employees (employee_id, first_name, last_name, email, hire_date, job_id)
  VALUES (v_emp_id, 'Jane', 'Smith', v_emp_email, SYSDATE, 'SA_REP');

  DBMS_OUTPUT.PUT_LINE('Inserted new employee with ID: ' || v_emp_id);

  COMMIT;
END;
```

Example 2: Using CURRVAL

```
DECLARE
  v_emp_id employees.employee_id%TYPE;
  v_dept_id departments.department_id%TYPE := 60;
BEGIN
  -- Insert a new employee
  INSERT INTO employees (employee_id, first_name, last_name, email, hire_date, job_id, department_id)
  VALUES (emp_id_seq.NEXTVAL, 'Robert', 'Johnson', 'RJOHNSON', SYSDATE, 'IT_PROG', v_dept_id);

  -- Get the current value of the sequence
  v_emp_id := emp_id_seq.CURRVAL;

  -- Use the CURRVAL to insert into another table
  INSERT INTO job_history (employee_id, start_date, end_date, job_id, department_id)
  VALUES (v_emp_id, SYSDATE, SYSDATE + 30, 'IT_PROG', v_dept_id);

  DBMS_OUTPUT.PUT_LINE('Inserted employee and job history for ID: ' || v_emp_id);

  COMMIT;
END;
```

These examples demonstrate the versatility and power of integrating SQL statements within PL/SQL blocks. They showcase how to retrieve data, manipulate records, and generate unique identifiers using sequences, all within the context of the HR schema. Practice these examples and try modifying them to deepen your understanding of SQL in PL/SQL.

Lab 3.2: Transaction Control in PL/SQL

After this lab, you will be able to:

- Use the COMMIT, ROLLBACK, and SAVEPOINT Statements
- Implement Error Handling with Transaction Control
- Use the SET TRANSACTION Statement
- Understand and Apply Transaction Isolation Levels

Transaction control in PL/SQL is crucial for maintaining data integrity and consistency. It allows you to group multiple SQL operations into a single, atomic unit of work.

COMMIT, ROLLBACK, and SAVEPOINT

These statements form the core of transaction control in PL/SQL.

Example 1: Basic Transaction Control

```
DECLARE
  v_emp_id employees.employee_id%TYPE := 103;
  v_salary_increase NUMBER := 1000;
BEGIN
  -- Start transaction
  UPDATE employees
  SET salary = salary + v_salary_increase
  WHERE employee_id = v_emp_id;

  DBMS_OUTPUT.PUT_LINE('Salary updated. Committing...');
  COMMIT;

  DBMS_OUTPUT.PUT_LINE('Transaction committed successfully.');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error occurred. Rolling back...');
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Transaction rolled back.');
END;
```

Example 2: Using SAVEPOINT

```
DECLARE
  v_dept_id departments.department_id%TYPE := 60;
  v_job_id jobs.job_id%TYPE := 'IT_PROG';
BEGIN
  -- Start transaction
  UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = v_dept_id;

  SAVEPOINT salary_update;

  UPDATE employees
  SET job_id = v_job_id
  WHERE department_id = v_dept_id;

  -- Simulate an error condition
  IF SQL%ROWCOUNT > 5 THEN
    DBMS_OUTPUT.PUT_LINE('Too many employees affected. Rolling back to SAVEPOINT...');
    ROLLBACK TO salary_update;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Changes acceptable. Committing...');
    COMMIT;
  END IF;

  DBMS_OUTPUT.PUT_LINE('Transaction completed.');
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    ROLLBACK;
END;
```

Implementing Error Handling with Transaction Control

Proper error handling is crucial when working with transactions.

Example 3: Advanced Error Handling

```
DECLARE
  e_too_many_updates EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_too_many_updates, -20001);

  v_dept_id departments.department_id%TYPE := 50;
  v_salary_increase NUMBER := 1.15; -- 15% increase
  v_affected_rows NUMBER;
BEGIN
  -- Start transaction
  UPDATE employees
  SET salary = salary * v_salary_increase
  WHERE department_id = v_dept_id;

  v_affected_rows := SQL%ROWCOUNT;

  IF v_affected_rows > 10 THEN
    RAISE e_too_many_updates;
  END IF;

  DBMS_OUTPUT.PUT_LINE('Salary updated for ' || v_affected_rows || ' employees.');
  COMMIT;

  DBMS_OUTPUT.PUT_LINE('Transaction committed successfully.');
EXCEPTION
  WHEN e_too_many_updates THEN
    DBMS_OUTPUT.PUT_LINE('Error: Too many employees affected (' || v_affected_rows || '). Rolling back...');
    ROLLBACK;
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM);
    ROLLBACK;
END;
```

SET TRANSACTION Statement

The SET TRANSACTION statement allows you to specify characteristics for the entire transaction.

Example 4: Read-Only Transaction

```
DECLARE
  v_total_salary NUMBER;
  v_emp_count NUMBER;
BEGIN
  COMMIT; -- End any previous transaction

  SET TRANSACTION READ ONLY NAME 'Salary Report';

  SELECT SUM(salary), COUNT(*)
  INTO v_total_salary, v_emp_count
  FROM employees;

  DBMS_OUTPUT.PUT_LINE('Total salary: $' || v_total_salary);
  DBMS_OUTPUT.PUT_LINE('Employee count: ' || v_emp_count);

  COMMIT; -- End the read-only transaction
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
    ROLLBACK;
END;
```

Understanding Transaction Isolation Levels

Oracle provides different isolation levels to control how transactions interact with each other.

Example 5: Serializable Transaction

```
DECLARE
  v_emp_count NUMBER;
BEGIN
  COMMIT; -- End any previous transaction

  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE NAME 'Emp Count';

  SELECT COUNT(*) INTO v_emp_count FROM employees;

  DBMS_OUTPUT.PUT_LINE('Employee count: ' || v_emp_count);

  -- Simulate some processing time
  DBMS_LOCK.SLEEP(5);

  -- Check if the count has changed
  SELECT COUNT(*) INTO v_emp_count FROM employees;

  DBMS_OUTPUT.PUT_LINE('Employee count after 5 seconds: ' || v_emp_count);

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

In this example, the SERIALIZABLE isolation level ensures that the transaction sees a consistent snapshot of the data, even if other sessions make changes.

Example 6: Read Committed vs. Serializable

To demonstrate the difference between READ COMMITTED (default) and SERIALIZABLE isolation levels, run these two scripts in separate sessions:

Session 1:

```
BEGIN SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
FOR i IN 1..3 LOOP
 SELECT COUNT(*) INTO v_count FROM employees;
 DBMS_OUTPUT.PUT_LINE('Count at ' || i || ' seconds: ' || v_count);
 DBMS_LOCK.SLEEP(1);
END LOOP;
COMMIT;
END;
```

Session 2:

```
BEGIN
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

  FOR i IN 1..3 LOOP
    SELECT COUNT(*) INTO v_count FROM employees;
    DBMS_OUTPUT.PUT_LINE('Count at ' || i || ' seconds: ' || v_count);
    DBMS_LOCK.SLEEP(1);
  END LOOP;

  COMMIT;
END;
```

While these scripts are running, execute an INSERT in a third session:

```
INSERT INTO employees (employee_id, first_name, last_name, email, hire_date, job_id)
VALUES (employees_seq.NEXTVAL, 'New', 'Employee', 'NEWEMP', SYSDATE, 'IT_PROG');
COMMIT;
```

You'll notice that the READ COMMITTED transaction might see the new employee, while the SERIALIZABLE transaction will not.

These examples demonstrate various aspects of transaction control in PL/SQL, including basic commit and rollback operations, savepoints, error handling, and transaction isolation levels. Understanding and correctly implementing these concepts is crucial for developing robust and reliable PL/SQL applications that maintain data integrity and consistency.

Summary

In this chapter, we've explored the powerful integration of SQL within PL/SQL, focusing on two main areas:

1. SQL Statements in PL/SQL
    - Initializing variables with SELECT INTO
    - Using DML statements (INSERT, UPDATE, DELETE) in PL/SQL blocks
    - Leveraging sequences for generating unique identifiers
2. Transaction Control in PL/SQL
    - Using COMMIT, ROLLBACK, and SAVEPOINT statements
    - Implementing error handling with transaction control
    - Utilizing the SET TRANSACTION statement
    - Understanding and applying transaction isolation levels

These concepts are fundamental to developing robust, efficient, and data-consistent PL/SQL applications. By mastering the use of SQL within PL/SQL, you can create more sophisticated database interactions, improve performance, and ensure data integrity.

Key takeaways include:

- The importance of proper error handling in transactional PL/SQL code

- The power of sequences for generating unique identifiers
- The significance of choosing the right transaction isolation level for your specific use case

As you continue to develop your PL/SQL skills, remember that the integration of SQL and PL/SQL is one of the most powerful features of Oracle's programming environment. It allows you to combine the declarative nature of SQL with the procedural capabilities of PL/SQL, resulting in highly efficient and maintainable database applications.

SQL IN PL/SQL