Chapter 12

Advanced Cursor Techniques

Mayko Freitas da Silva

Introduction

Welcome back to our journey through the world of PL/SQL cursors. In the previous chapter, you learned the basics of cursors, which are like bookmarks in a database, helping you navigate through sets of data. Now, we're going to explore more advanced techniques that will make your cursors even more powerful and flexible.

Imagine you're in a vast library of information. In the last chapter, you learned how to find and read individual books (basic cursors). Now, you're going to learn how to create customizable reading lists (parameterized cursors), use adaptable bookmarks that can point to different sections in various books (cursor variables), work with books that contain references to other books (cursor expressions), and even how to temporarily reserve books for editing (FOR UPDATE cursors).

In this chapter, you will learn about:

1. Enhancing cursors with parameters
2. Using flexible cursor variables
3. Working with cursor expressions and nested data
4. Updating data with FOR UPDATE cursors

These advanced techniques will allow you to create more dynamic and adaptable database operations, much like a skilled librarian who can effortlessly navigate and manage complex collections of information.

Let's break down what you'll be learning:

1. **Enhancing cursors with parameters**: This is like creating a customizable reading list. You'll learn how to make your cursors more flexible by adding parameters, allowing you to reuse the same cursor for different data subsets.
2. **Using flexible cursor variables**: Think of cursor variables as adaptable bookmarks that can be moved between different books. You'll discover how to create cursor variables that can be associated with different queries at runtime, providing incredible flexibility in your code.
3. **Working with cursor expressions and nested data**: This is similar to working with books that contain references to other books. You'll explore how to handle more intricate data structures using cursor expressions and nested cursors.
4. **Updating data with FOR UPDATE cursors**: Lastly, you'll learn about FOR UPDATE cursors, which are like placing a "reserved for editing" slip in a book. This technique allows you to lock specific rows for updating, ensuring data integrity in multi-user environments.

By mastering these advanced cursor techniques, you'll be able to write more efficient, flexible, and powerful PL/SQL code. You'll have the tools to handle complex data retrieval and manipulation tasks with ease, making you a true expert in navigating and managing your database library.

Are you ready to expand your cursor toolkit and become a master librarian of your database? Let's dive in and start exploring these advanced techniques.

Lab 12.1: Enhancing Cursors with Parameters

After this lab, you will be able to:

- Declare and use parameterized cursors
- Implement default parameter values
- Pass variables to cursor parameters

Introduction to Parameterized Cursors

Parameterized cursors are like customizable reading lists in our database library. They allow you to create a single cursor that can be used to retrieve different sets of data based on the parameters you provide. This flexibility makes your code more reusable and efficient.

Declaring Parameterized Cursors

To declare a parameterized cursor, you include the parameter list in the cursor declaration. Here's the general syntax:

```
CURSOR cursor_name (parameter_name datatype, ...) IS
  SELECT statement;
```

Let's see an example using the HR schema:

```
DECLARE
 CURSOR c_employees (p_department_id NUMBER) IS
  SELECT employee_id, first_name, last_name
  FROM employees
  WHERE department_id = p_department_id;

 v_emp_id employees.employee_id%TYPE;
 v_first_name employees.first_name%TYPE;
 v_last_name employees.last_name%TYPE;
BEGIN
 OPEN c_employees(60);  -- Open cursor for department 60

 LOOP
  FETCH c_employees INTO v_emp_id, v_first_name, v_last_name;
  EXIT WHEN c_employees%NOTFOUND;

  DBMS_OUTPUT.PUT_LINE('Employee: ' || v_first_name || ' ' || v_last_name);
 END LOOP;

 CLOSE c_employees;
END;
```

In this example, c_employees is a parameterized cursor that takes a department ID as a parameter. This allows you to use the same cursor to retrieve employees from any department.

Using Default Parameter Values

You can make your parameterized cursors even more flexible by providing default values for parameters. This is like having a default reading list that can be customized if needed.

```
DECLARE
  CURSOR c_employees (p_department_id NUMBER DEFAULT 60) IS
    SELECT employee_id, first_name, last_name
    FROM employees
    WHERE department_id = p_department_id;

  v_emp_id employees.employee_id%TYPE;
  v_first_name employees.first_name%TYPE;
  v_last_name employees.last_name%TYPE;
BEGIN
  -- Open cursor with default parameter
  OPEN c_employees;

  LOOP
    FETCH c_employees INTO v_emp_id, v_first_name, v_last_name;
    EXIT WHEN c_employees%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_first_name || ' ' || v_last_name);
  END LOOP;

  CLOSE c_employees;

  -- Open cursor with a different parameter
  OPEN c_employees(30);

  LOOP
    FETCH c_employees INTO v_emp_id, v_first_name, v_last_name;
    EXIT WHEN c_employees%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_first_name || ' ' || v_last_name);
  END LOOP;

  CLOSE c_employees;
END;
```

In this example, the cursor c_employees has a default parameter value of 60. You can open it without specifying a parameter to use the default, or provide a different value to override the default.

Passing Variables to Cursor Parameters

You can also pass variables to cursor parameters, allowing for even more dynamic behavior:

```
DECLARE
  CURSOR c_employees (p_department_id NUMBER) IS
    SELECT employee_id, first_name, last_name
    FROM employees
    WHERE department_id = p_department_id;

  v_emp_id employees.employee_id%TYPE;
  v_first_name employees.first_name%TYPE;
  v_last_name employees.last_name%TYPE;
  v_dept_id employees.department_id%TYPE := 50;
BEGIN
  OPEN c_employees(v_dept_id);

  LOOP
    FETCH c_employees INTO v_emp_id, v_first_name, v_last_name;
    EXIT WHEN c_employees%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_first_name || ' ' || v_last_name);
  END LOOP;

  CLOSE c_employees;
END;
```

In this example, we're passing the variable v_dept_id to the cursor. This allows you to change the department ID dynamically in your code.

Exercise 12.1

Create a parameterized cursor that retrieves employees based on their job title. The cursor should accept two parameters: the job title and a salary threshold. It should return the employee's name, job title, and salary for all employees with the specified job title and a salary greater than the threshold. Use default values for both parameters. Test your cursor with different job titles and salary thresholds.

By mastering parameterized cursors, you've added a powerful tool to your PL/SQL toolkit. These flexible cursors allow you to write more reusable and efficient code, adapting to different data retrieval needs with ease. In the next lab, we'll explore the concept of cursor variables, which provide even more flexibility in working with cursors.

This lab section introduces parameterized cursors, explains how to declare and use them, and provides examples using the HR schema. It also includes an exercise for practice.

Lab 12.2: Exploring Cursor Variables and Expressions

After this lab, you will be able to:

- Declare and use cursor variables
- Distinguish between strong and weak REF CURSOR types
- Process cursor variables using OPEN FOR, FETCH, and CLOSE
- Work with cursor expressions and nested cursors

Introduction to Cursor Variables

Cursor variables are like adaptable bookmarks in our database library. Unlike regular cursors that are tied to a specific query, cursor variables can be associated with different queries at runtime. This flexibility allows you to create more dynamic and reusable code.

Declaring Cursor Variables

There are two types of cursor variables:

1. Strong REF CURSOR: Tied to a specific return type

2. Weak REF CURSOR: Can be associated with any query

Here's how to declare both types:

```
-- Strong REF CURSOR
TYPE emp_cursor_type IS REF CURSOR RETURN employees%ROWTYPE;

-- Weak REF CURSOR
TYPE generic_cursor_type IS REF CURSOR;

-- Predefined weak REF CURSOR
SYS_REFCURSOR;
```

Let's see an example using a weak REF CURSOR:

```
DECLARE
  TYPE generic_cursor_type IS REF CURSOR;
  v_cursor generic_cursor_type;
  v_employee employees%ROWTYPE;
BEGIN
  OPEN v_cursor FOR
    SELECT * FROM employees WHERE department_id = 60;

  LOOP
    FETCH v_cursor INTO v_employee;
    EXIT WHEN v_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_employee.first_name || ' ' || v_employee.last_name);
  END LOOP;

  CLOSE v_cursor;
END;
```

In this example, we're using a weak REF CURSOR that can be associated with any query.

Strong vs. Weak REF CURSOR Types

Strong REF CURSORs are type-safe but less flexible:

```
DECLARE
  TYPE emp_cursor_type IS REF CURSOR RETURN employees%ROWTYPE;
  v_emp_cursor emp_cursor_type;
  v_employee employees%ROWTYPE;
BEGIN
  OPEN v_emp_cursor FOR
    SELECT * FROM employees WHERE department_id = 60;

  LOOP
    FETCH v_emp_cursor INTO v_employee;
    EXIT WHEN v_emp_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_employee.first_name || ' ' || v_employee.last_name);
  END LOOP;

  CLOSE v_emp_cursor;

  -- This would cause an error because the return type doesn't match
  -- OPEN v_emp_cursor FOR
  --   SELECT * FROM departments;
END;
```

Weak REF CURSORs are more flexible but require careful handling:

```
DECLARE
  TYPE generic_cursor_type IS REF CURSOR;
  v_cursor generic_cursor_type;
  v_employee employees%ROWTYPE;
  v_department departments%ROWTYPE;
BEGIN
  -- Using the cursor for employees
  OPEN v_cursor FOR
    SELECT * FROM employees WHERE department_id = 60;

  LOOP
    FETCH v_cursor INTO v_employee;
    EXIT WHEN v_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_employee.first_name || ' ' || v_employee.last_name);
  END LOOP;

  CLOSE v_cursor;

  -- Using the same cursor for departments
  OPEN v_cursor FOR
    SELECT * FROM departments WHERE department_id = 60;

  LOOP
    FETCH v_cursor INTO v_department;
    EXIT WHEN v_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Department: ' || v_department.department_name);
  END LOOP;

  CLOSE v_cursor;
END;
```

Cursor Expressions

Cursor expressions allow you to use cursors in SQL statements. They're like creating a temporary reading list within a larger query. Here's an example:

```
SELECT d.department_name,
      CURSOR(SELECT e.first_name, e.last_name
           FROM employees e
           WHERE e.department_id = d.department_id) as emp_list
FROM departments d
WHERE d.department_id = 60;
```

To use this in PL/SQL:

```
DECLARE
 TYPE dept_cursor_type IS REF CURSOR;
 v_dept_cursor dept_cursor_type;
 v_dept_name departments.department_name%TYPE;
 v_emp_cursor SYS_REFCURSOR;
 v_first_name employees.first_name%TYPE;
 v_last_name employees.last_name%TYPE;
BEGIN
 OPEN v_dept_cursor FOR
  SELECT d.department_name,
       CURSOR(SELECT e.first_name, e.last_name
            FROM employees e
            WHERE e.department_id = d.department_id) as emp_list
  FROM departments d
  WHERE d.department_id = 60;

 FETCH v_dept_cursor INTO v_dept_name, v_emp_cursor;

 DBMS_OUTPUT.PUT_LINE('Department: ' || v_dept_name);

 LOOP
  FETCH v_emp_cursor INTO v_first_name, v_last_name;
  EXIT WHEN v_emp_cursor%NOTFOUND;

  DBMS_OUTPUT.PUT_LINE('  Employee: ' || v_first_name || ' ' || v_last_name);
 END LOOP;

 CLOSE v_dept_cursor;
END;
```

This example demonstrates how to use a cursor expression to create a nested cursor, allowing you to fetch related data in a hierarchical manner.

Exercise 12.2

Create a PL/SQL block that uses a cursor variable to dynamically switch between querying employees and departments. The block should accept user input to determine which table to query. Use a weak REF CURSOR to accomplish this. Then, extend your solution to include a cursor expression that returns a list of employees for each department when querying departments.

By mastering cursor variables and expressions, you've gained powerful tools for creating flexible and dynamic database queries. These techniques allow you to write more adaptable code that can handle a variety of data retrieval scenarios. In the next lab, we'll explore how to use cursors for updating data with the FOR UPDATE clause.

Lab 12.3: Updating Data with FOR UPDATE Cursors

After this lab, you will be able to:

- Understand the purpose and use of FOR UPDATE cursors
- Implement SELECT FOR UPDATE statements
- Use the WHERE CURRENT OF clause for precise updates
- Recognize potential issues with FOR UPDATE cursors and how to avoid them

Introduction to FOR UPDATE Cursors

FOR UPDATE cursors are like placing a "reserved for editing" slip in a book in our database library. They allow you to lock specific rows that you intend to update, preventing other sessions from modifying these rows until your transaction is complete. This ensures data consistency in multi-user environments.

Using SELECT FOR UPDATE Statements

To create a FOR UPDATE cursor, you add the FOR UPDATE clause to your SELECT statement. Here's the basic syntax:

```
CURSOR cursor_name IS
  SELECT column1, column2, ...
  FROM table_name
  WHERE condition
  FOR UPDATE [OF column1, column2, ...] [NOWAIT | WAIT n];
```

Let's see an example using the HR schema:

```
DECLARE
  CURSOR c_emp_update IS
    SELECT employee_id, salary
    FROM employees
    WHERE department_id = 60
    FOR UPDATE OF salary NOWAIT;

  v_emp_id employees.employee_id%TYPE;
  v_salary employees.salary%TYPE;
BEGIN
  FOR emp_rec IN c_emp_update
  LOOP
    -- Increase salary by 10%
    UPDATE employees
    SET salary = emp_rec.salary * 1.1
    WHERE CURRENT OF c_emp_update;

    DBMS_OUTPUT.PUT_LINE('Updated salary for employee ' || emp_rec.employee_id);
  END LOOP;

  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END;
```

In this example, we're creating a FOR UPDATE cursor that locks the salary column for employees in department 60. The NOWAIT option means the cursor will raise an exception if it can't immediately acquire the lock.

Using the WHERE CURRENT OF Clause

The WHERE CURRENT OF clause allows you to update or delete the current row being processed by the cursor. This ensures that you're modifying exactly the row you intend to, even if the data has changed since you opened the cursor.

```
DECLARE
  CURSOR c_emp_update IS
    SELECT employee_id, salary
    FROM employees
    WHERE department_id = 50
    FOR UPDATE;

  v_salary_increase NUMBER := 1000;
BEGIN
  FOR emp_rec IN c_emp_update
  LOOP
    IF emp_rec.salary < 5000 THEN
      UPDATE employees
      SET salary = salary + v_salary_increase
      WHERE CURRENT OF c_emp_update;

      DBMS_OUTPUT.PUT_LINE('Updated salary for employee ' || emp_rec.employee_id);
    END IF;
  END LOOP;

  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
END;
```

In this example, we're using WHERE CURRENT OF to ensure we're updating the exact row currently being processed by the cursor.

Potential Issues and Best Practices

1. **Lock Contention**: FOR UPDATE cursors can lead to lock contention in busy systems. Use them judiciously and consider the NOWAIT or WAIT n options to avoid long-running transactions.
2. **Deadlocks**: Be cautious when using multiple FOR UPDATE cursors to avoid deadlocks. Process rows in a consistent order across your application.
3. **Performance**: FOR UPDATE locks can impact performance. Only lock the columns you actually need to update.
4. **Commit Frequency**: In long-running operations, consider committing in batches to release locks and reduce contention.

Here's an example demonstrating batch processing:

```
DECLARE
  CURSOR c_emp_update IS
    SELECT employee_id, salary
    FROM employees
    WHERE department_id = 50
    FOR UPDATE;

  v_counter NUMBER := 0;
  v_batch_size NUMBER := 10;
BEGIN
  FOR emp_rec IN c_emp_update
  LOOP
    -- Debugging output to see which employee is being updated
    DBMS_OUTPUT.PUT_LINE('Updating employee_id: ' || emp_rec.employee_id || ' with current salary: ' || emp_r
ec.salary);

    UPDATE employees
    SET salary = salary * 1.05
    WHERE CURRENT OF c_emp_update;

    v_counter := v_counter + 1;

    -- Commit after processing all records, not inside the loop
    IF v_counter MOD v_batch_size = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Processed batch of ' || v_batch_size || ' updates');
    END IF;
  END LOOP;

  COMMIT;  -- Commit all updates after the loop
  DBMS_OUTPUT.PUT_LINE('Final commit. Total updates: ' || v_counter);
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Error occurred: ' || SQLERRM);
    RAISE;
END;
```

This script processes employees in batches, committing every 10 updates to release locks and reduce the transaction size.

Exercise 12.3

Create a PL/SQL block that uses a FOR UPDATE cursor to process salary increases for employees. The cursor should select employees from a specific department (you choose) and lock their salary and commission_pct columns. For each employee, if their salary is below the average for their job_id, increase it by 5%. If they don't have a commission, set it to 0.1. Use error handling to manage potential lock timeouts, and implement batch processing to commit every 5 employees. Test your solution and observe how it handles concurrent access.

By mastering FOR UPDATE cursors, you've learned how to safely update data in multi-user environments, ensuring data consistency and integrity. Remember to use these cursors judiciously and always consider their impact on system performance and concurrency. In the next and final section, we'll summarize what we've learned and discuss how these advanced cursor techniques can be applied in real-world scenarios.

Summary

In this chapter, we've explored advanced cursor techniques that enhance your ability to work with data in PL/SQL. Let's recap the key concepts we've covered:

1. **Parameterized Cursors**
    - Allow you to create flexible, reusable cursors by accepting parameters
    - Enable dynamic data retrieval based on input values
    - Can have default parameter values for added flexibility
2. **Cursor Variables**
    - Provide runtime flexibility by allowing association with different queries
    - Come in two types: strong (type-safe) and weak (flexible) REF CURSORs
    - Enable more dynamic and adaptable code structures
3. **Cursor Expressions**
    - Allow the use of cursors within SQL statements
    - Facilitate the creation of nested cursors for hierarchical data retrieval
    - Enhance the ability to work with complex, related datasets
4. **FOR UPDATE Cursors**
    - Enable row-level locking for data consistency in multi-user environments
    - Use the WHERE CURRENT OF clause for precise updates on the current row
    - Require careful management to avoid lock contention and deadlocks

These advanced techniques significantly expand your toolkit for working with data in Oracle databases. Here's how they can be applied in real-world scenarios:

- **Parameterized Cursors**: Ideal for creating generic data retrieval routines that can be easily customized, such as generating reports for different departments or time periods.
- **Cursor Variables**: Excellent for building flexible data access layers in applications, allowing the same code to handle various types of queries based on user input or application state.
- **Cursor Expressions**: Powerful for working with hierarchical or related data, such as retrieving employee information along with their project assignments or subordinates.
- **FOR UPDATE Cursors**: Essential for implementing safe data modification processes in multi-user systems, such as updating inventory levels or processing customer orders.

As you apply these techniques, remember these best practices:

1. Always consider performance implications, especially with nested cursors or FOR UPDATE clauses.
2. Use strong REF CURSORs when type safety is crucial, and weak REF CURSORs when flexibility is needed.
3. Implement proper error handling, particularly when dealing with FOR UPDATE cursors that may encounter locked rows.
4. Be mindful of transaction management, committing or rolling back changes appropriately to maintain data consistency.

By mastering these advanced cursor techniques, you've significantly enhanced your ability to create efficient, flexible, and robust PL/SQL code. You're now equipped to handle complex data retrieval and manipulation tasks with ease, making you a more effective database developer.

As you continue to develop your PL/SQL skills, look for opportunities to apply these techniques in your projects. Practice combining different cursor types and experiment with various scenarios to deepen your understanding. Remember, the key to mastery is not just knowing these techniques, but understanding when and how to apply them effectively in real-world situations.

Advanced Cursor Techniques