Chapter 7

Mastering Loop Control in PL/SQL

Mayko Freitas da Silva

Introduction

Welcome back to your PL/SQL journey! In our previous chapter, you learned about the basics of loops - those powerful tools that help us repeat actions in our code. Now, it's time to take your loop skills to the next level.

Imagine you're following a recipe to bake cookies. Sometimes, you might want to skip a step or repeat a set of steps multiple times. In PL/SQL, we have special tools to do just that with our code: CONTINUE statements and nested loops.

In this chapter, you will learn about:

- CONTINUE Statements
- Nested Loops

These advanced loop control techniques will give you more power over how your code flows, making your programs more efficient and flexible.

Think of CONTINUE statements as a way to say, "Let's skip this part and move on to the next item in our list." It's like deciding to skip adding chocolate chips to some cookies in your batch.

Nested loops, on the other hand, are like having a smaller loop inside a bigger one. Imagine you're making different types of cookies, and for each type, you're repeating the steps of shaping and baking individual cookies. That's what nested loops do in code!

Are you ready to enhance your PL/SQL recipe book with these new techniques? Let's dive in and start with our first lab on CONTINUE statements.

Certainly! Let's move on to the first lab, focusing on CONTINUE statements without using cursors.

Lab 7.1: Skipping Ahead with CONTINUE

After this lab, you will be able to:

- Use CONTINUE Statements
- Use CONTINUE WHEN Statements

The Basic CONTINUE Statement

The CONTINUE statement is like a "skip to the next item" button in your loop. When PL/SQL encounters a CONTINUE, it immediately jumps to the next iteration of the loop, skipping any remaining code in the current iteration.

Let's look at an example:

```
DECLARE
  v_counter NUMBER := 0;
  v_sum NUMBER := 0;
BEGIN
  LOOP
    v_counter := v_counter + 1;

    -- Skip even numbers
    IF MOD(v_counter, 2) = 0 THEN
      CONTINUE;
    END IF;

    v_sum := v_sum + v_counter;
    DBMS_OUTPUT.PUT_LINE('Added: ' || v_counter);

    EXIT WHEN v_counter >= 10;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Sum of odd numbers: ' || v_sum);
END;
```

In this example, we're adding up odd numbers from 1 to 10. The CONTINUE statement helps us skip even numbers. It's like having a cookie recipe where you decide to skip every other step.

CONTINUE WHEN: Conditional Skipping

The CONTINUE WHEN statement combines the IF and CONTINUE into one smooth move. It's a shortcut for conditional skipping.

Here's how you might use it:

```
DECLARE
  v_counter NUMBER := 0;
  v_sum NUMBER := 0;
BEGIN
  LOOP
    v_counter := v_counter + 1;

    -- Skip numbers not divisible by 3
    CONTINUE WHEN MOD(v_counter, 3) != 0;

    v_sum := v_sum + v_counter;
    DBMS_OUTPUT.PUT_LINE('Added: ' || v_counter);

    EXIT WHEN v_counter >= 30;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Sum of numbers divisible by 3: ' || v_sum);
END;
```

In this case, we're summing up numbers divisible by 3 from 1 to 30. The CONTINUE WHEN statement makes our code cleaner and more concise.

Placement Matters: Flow Control in Loops

Where you put your CONTINUE statement can make a big difference. It's like deciding when to skip a step in your recipe - the timing matters!

Let's look at an example:

```
DECLARE
  v_counter NUMBER := 0;
  v_sum_odd NUMBER := 0;
  v_sum_even NUMBER := 0;
BEGIN
  LOOP
    v_counter := v_counter + 1;

    -- Exit condition
    EXIT WHEN v_counter > 20;

    -- Continue condition
    CONTINUE WHEN MOD(v_counter, 2) = 0;

    v_sum_odd := v_sum_odd + v_counter;
    DBMS_OUTPUT.PUT_LINE('Added to odd sum: ' || v_counter);

    CONTINUE;

    -- This code will never execute due to the CONTINUE above
    v_sum_even := v_sum_even + v_counter;
    DBMS_OUTPUT.PUT_LINE('Added to even sum: ' || v_counter);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Sum of odd numbers: ' || v_sum_odd);
  DBMS_OUTPUT.PUT_LINE('Sum of even numbers: ' || v_sum_even);
END;
```

In this example, we have multiple CONTINUE statements. The first one skips even numbers, and the second one prevents the code for even numbers from ever executing. It's like having a recipe where you decide to skip certain steps entirely for some cookies in your batch.

Remember, the order of your conditions and CONTINUE statements can greatly affect how your loop behaves. It's crucial to think carefully about where you place them in your code.

Now that you've seen how CONTINUE statements work, try experimenting with them in your own PL/SQL blocks. In our next lab, we'll explore how to nest these loops to handle more complex scenarios.

Certainly! Let's move on to the next section, which covers nested loops.

Lab 7.2: Loops within Loops

After this lab, you will be able to:

- Use Nested Loops
- Use Loop Labels

Nesting Different Loop Types

Nested loops are like having a smaller loop inside a bigger loop. Imagine you're organizing your closet: the outer loop might be going through each shelf, while the inner loop sorts items on that shelf.

Let's look at an example using nested loops:

```
DECLARE
  v_outer NUMBER := 0;
  v_inner NUMBER := 0;
  v_result NUMBER := 0;
BEGIN
  LOOP
    v_outer := v_outer + 1;
    DBMS_OUTPUT.PUT_LINE('Outer loop iteration: ' || v_outer);

    v_inner := 0;
    WHILE v_inner < 3 LOOP
      v_inner := v_inner + 1;
      v_result := v_outer * v_inner;
      DBMS_OUTPUT.PUT_LINE('  Inner loop iteration: ' || v_inner || ', Result: ' || v_result);
    END LOOP;

    EXIT WHEN v_outer = 3;
  END LOOP;
END;
```

In this example, we have an outer LOOP that runs three times, and for each iteration, an inner WHILE loop runs three times. It's like going through three shelves in your closet, and for each shelf, you're organizing three items.

Keeping Track with Loop Labels

When you're dealing with nested loops, it can sometimes get confusing to keep track of which loop is which. That's where loop labels come in handy. They're like putting clear labels on each shelf in your closet.

Here's how you can use loop labels:

```
DECLARE
  v_outer NUMBER := 0;
  v_inner NUMBER := 0;
  v_sum NUMBER := 0;
BEGIN
  <<outer_loop>>
  LOOP
    v_outer := v_outer + 1;
    DBMS_OUTPUT.PUT_LINE('Outer loop iteration: ' || v_outer);

    <<inner_loop>>
    LOOP
      v_inner := v_inner + 1;
      v_sum := v_sum + v_inner;

      EXIT inner_loop WHEN v_inner = v_outer;
    END LOOP inner_loop;

    DBMS_OUTPUT.PUT_LINE('  Sum for this outer iteration: ' || v_sum);
    v_inner := 0;
    v_sum := 0;

    EXIT outer_loop WHEN v_outer = 5;
  END LOOP outer_loop;
END;
```

In this example, we've labeled our loops as outer_loop and inner_loop. Notice how we use these labels with EXIT statements to specify which loop we're exiting. It's like being able to say "I'm done with this shelf" or "I'm done with the whole closet."

Navigating Multiple Loops

Sometimes, you might need more complex logic involving multiple nested loops. Let's look at an example:

```
DECLARE
  v_outer NUMBER := 0;
  v_middle NUMBER := 0;
  v_inner NUMBER := 0;
  v_result NUMBER := 0;
BEGIN
  <<outer_loop>>
  FOR v_outer IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE('Outer loop: ' || v_outer);

    <<middle_loop>>
    FOR v_middle IN 1..2 LOOP
      DBMS_OUTPUT.PUT_LINE('  Middle loop: ' || v_middle);

      <<inner_loop>>
      FOR v_inner IN 1..4 LOOP
        CONTINUE outer_loop WHEN v_inner = 3;

        v_result := v_outer * v_middle * v_inner;
        DBMS_OUTPUT.PUT_LINE('    Inner loop: ' || v_inner || ', Result: ' || v_result);
      END LOOP inner_loop;
    END LOOP middle_loop;
  END LOOP outer_loop;
END;
```

In this example, we have three nested loops. The CONTINUE statement in the innermost loop actually skips to the next iteration of the outermost loop when v_inner is 3. It's like deciding to move to the next shelf entirely if you find a certain type of item while organizing.

Summary

In this chapter, you've learned how to fine-tune your loops using CONTINUE statements and how to create more complex loop structures with nested loops. You've seen how CONTINUE can help you skip iterations efficiently, and how nested loops allow you to process multi-level data effectively. Finally, you've discovered how loop labels can make your code more readable and give you more control in complex loop structures.

These tools will allow you to write more sophisticated and efficient PL/SQL code, handling complex data processing tasks with ease. Remember, like organizing a complex closet system, the key is knowing when and how to use each of these techniques to create smooth, efficient code.

Practice Exercises

1. Write a PL/SQL block that uses a CONTINUE statement to skip processing for numbers divisible by 5 in a loop from 1 to 20.
2. Create a nested loop structure that prints a multiplication table for numbers 1 to 5, but use a CONTINUE WHEN statement to skip the product when it's greater than 15.
3. Use loop labels to create a pattern of asterisks that forms a right-angled triangle, where the outer loop controls the rows and the inner loop controls the columns. Exit the entire structure when the triangle has 5 rows.

Mastering Loop Control in PL/SQL