Chapter 4

Mastering Conditional Logic in PL/SQL

Mayko Freitas da Silva

Introduction

Imagine you're a chef in a bustling kitchen. As you prepare dishes, you constantly make decisions based on various factors: Is the pasta al dente? Has the sauce reduced enough? Is the steak medium-rare? In the world of PL/SQL programming, you'll face similar decision-making scenarios, but instead of ingredients and cooking times, you'll be working with data and conditions.

Conditional logic in PL/SQL is your set of kitchen tools that help you make these important decisions in your code. Just as a chef uses different utensils for different tasks, you'll use various conditional statements to control the flow of your program based on specific conditions.

In this chapter, you will learn about:

1. IF Statements: Your basic kitchen knife, essential for simple decision-making.
2. ELSIF Statements: Think of these as your set of specialized knives, allowing you to handle more complex scenarios.
3. Nested IF Statements: These are like your nested mixing bowls, enabling you to create layered decisions within decisions.
4. Logical Operators: Consider these your spices, allowing you to combine and enhance your conditions for more nuanced control.

By the end of this chapter, you'll be equipped to "cook up" sophisticated PL/SQL programs that can make intelligent decisions based on various conditions, just like a master chef creating culinary masterpieces.

Let's start our journey into the world of conditional logic in PL/SQL, where you'll learn to write code that can adapt and respond to different situations, making your database applications more intelligent and efficient.

Lab 4.1: Understanding IF Statements

After this lab, you will be able to:

- Use IF-THEN Statements
- Implement IF-THEN-ELSE Statements
- Handle NULL conditions effectively

Imagine you're a bouncer at a club. Your job is to make decisions about who can enter based on certain criteria. This is exactly what IF statements do in PL/SQL – they allow your code to make decisions based on specific conditions.

IF-THEN Structure: The Basic Bouncer

The IF-THEN statement is like a bouncer with one simple rule: "If you're 21 or older, you can enter." Here's how it looks in PL/SQL:

```
IF condition THEN
   statement1;
   statement2;
   ...
END IF;
```

Let's see this in action with the HR schema:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_salary NUMBER;
BEGIN
   SELECT salary INTO v_salary
   FROM employees
   WHERE employee_id = v_employee_id;

   IF v_salary > 10000 THEN
      DBMS_OUTPUT.PUT_LINE('VIP Guest: Employee ' || v_employee_id || ' earns over $10,000');
   END IF;
END;
```

In this example, you're checking if an employee is a "VIP" (salary over $10,000). If they are, you announce their VIP status.

IF-THEN-ELSE Structure: The Thoughtful Bouncer

Now, let's make our bouncer more communicative. The IF-THEN-ELSE structure is like a bouncer who not only lets people in but also politely informs those who can't enter. Here's the structure:

```
IF condition THEN
   statement1;
ELSE
   statement2;
END IF;
```

Let's modify our previous example:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_salary NUMBER;
BEGIN
   SELECT salary INTO v_salary
   FROM employees
   WHERE employee_id = v_employee_id;

   IF v_salary > 10000 THEN
      DBMS_OUTPUT.PUT_LINE('VIP Guest: Employee ' || v_employee_id || ' earns over $10,000');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Regular Guest: Employee ' || v_employee_id || ' earns $10,000 or less');
   END IF;
END;
```

Now, you have a message for both VIP and regular employees.

Handling NULL Conditions: The Cautious Bouncer

In PL/SQL, NULL is like a mysterious guest with no clear information. A cautious bouncer needs to handle these cases specially. When dealing with NULL in IF statements, remember that any comparison with NULL results in NULL, not TRUE or FALSE.

Let's see how to handle this:

```
DECLARE
   v_commission_pct employees.commission_pct%TYPE;
   v_employee_id NUMBER := 145;
BEGIN
   SELECT commission_pct INTO v_commission_pct
   FROM employees
   WHERE employee_id = v_employee_id;

   IF v_commission_pct IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ' does not earn commission.');
   ELSIF v_commission_pct > 0 THEN
       DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ' earns commission: ' || v_commission_pct*100 ||
'%');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ' has 0% commission.');
   END IF;
END;
```

In this example, you're checking if an employee earns commission. The IS NULL condition is used to properly handle NULL values, like a cautious bouncer dealing with guests who have unclear information.

Practice Exercise

Now it's your turn to be the bouncer! Write a PL/SQL block that checks an employee's job title (job_id in the employees table). If the job_id contains 'MANAGER', print "Welcome, Manager!". Otherwise, print "Welcome, Team Member!". Use employee_id 101 for this exercise.

Remember, just like a good bouncer needs practice to make quick and accurate decisions, you'll get better at using IF statements with practice. Try modifying the conditions and messages to see how the behavior changes!

Certainly! Here's a more didactic approach to the "Lab 4.2: Working with ELSIF Statements" session:

Lab 4.2: Working with ELSIF Statements

After this lab, you will be able to:

- Use ELSIF Statements effectively
- Handle multiple conditions in a structured manner

Imagine you're a hotel receptionist assigning rooms based on guest preferences. Some guests want ocean views, others prefer mountain views, and some prioritize proximity to the elevator. This scenario is perfect for understanding ELSIF statements in PL/SQL.

The ELSIF Structure: The Versatile Receptionist

ELSIF statements allow you to check multiple conditions in a structured way, much like a receptionist working through a list of room preferences. Here's the structure:

```
IF condition1 THEN
   statement1;
ELSIF condition2 THEN
   statement2;
ELSIF condition3 THEN
   statement3;
ELSE
   statement4;
END IF;
```

Let's see this in action with the HR schema, categorizing employees based on their salary:

```
DECLARE
  v_employee_id NUMBER := 101;
  v_salary NUMBER;
BEGIN
  SELECT salary INTO v_salary
  FROM employees
  WHERE employee_id = v_employee_id;

  IF v_salary < 5000 THEN
    DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ': Economy Class');
  ELSIF v_salary BETWEEN 5000 AND 10000 THEN
    DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ': Business Class');
  ELSIF v_salary > 10000 THEN
    DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ': First Class');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ': Salary information unavailable');
  END IF;
END;
```

Important Points to Remember

1. **Order Matters**: ELSIF conditions are evaluated in order. Once a condition is true, the corresponding block is executed, and the rest are skipped. It's like the receptionist stopping once they find a suitable room.
2. **ELSE is Optional**: You don't always need an ELSE clause, but it's good practice to include one to handle unexpected cases.
3. **Multiple ELSIFs**: You can have as many ELSIF clauses as you need, but too many can make your code hard to read.

Practical Example: Employee Performance Categorization

Let's create a more complex example using multiple criteria from the HR schema:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_salary NUMBER;
   v_commission_pct NUMBER;
   v_department_id NUMBER;
BEGIN
   SELECT salary, commission_pct, department_id
   INTO v_salary, v_commission_pct, v_department_id
   FROM employees
   WHERE employee_id = v_employee_id;

   IF v_salary > 10000 AND v_commission_pct IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ': Top Performer');
   ELSIF v_salary > 8000 OR v_department_id = 80 THEN -- 80 is Sales department
      DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ': High Potential');
   ELSIF v_salary > 5000 THEN
      DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ': Solid Contributor');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee ' || v_employee_id || ': Needs Evaluation');
   END IF;
END;
```

This script categorizes employees based on multiple factors: salary, commission, and department. It's like a receptionist considering multiple guest preferences to find the perfect room.

Practice Exercise

Now it's your turn to be the receptionist! Write a PL/SQL block that categorizes employees based on their job_id and salary. Use the following criteria:

- If job_id contains 'MANAGER' and salary > 10000: "Senior Manager"
- If job_id contains 'MANAGER' and salary <= 10000: "Junior Manager"
- If job_id contains 'REP' and salary > 8000: "Senior Sales Representative"
- If job_id contains 'REP' and salary <= 8000: "Junior Sales Representative"
- For all others: "Staff Member"

Use employee_id 145 for this exercise.

Remember, just like a skilled receptionist can handle various guest preferences smoothly, with practice, you'll become proficient at using ELSIF statements to handle complex decision-making in your PL/SQL code!

Lab 4.3: Mastering Nested IF Statements

After this lab, you will be able to:

- Implement Nested IF Statements
- Understand the concept of inner and outer IF statements
- Create complex decision structures in PL/SQL

Imagine you're a travel agent planning a vacation package. You first decide on the destination (beach or mountains), then choose activities (swimming or hiking), and finally select accommodations (hotel or cabin). This multi-level decision-making process is exactly what nested IF statements allow you to do in PL/SQL.

Understanding Nested IF Statements: The Travel Package Planner

Nested IF statements are like planning a customized travel package. You make one decision, then based on that, you make another, and so on. Here's a basic structure:

```
IF outer_condition THEN
   IF inner_condition THEN
      inner_statement;
   ELSE
      alternative_inner_statement;
   END IF;
ELSE
   outer_alternative_statement;
END IF;
```

Let's see this in action using the HR schema:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_salary NUMBER;
   v_department_id NUMBER;
BEGIN
   SELECT salary, department_id INTO v_salary, v_department_id
   FROM employees
   WHERE employee_id = v_employee_id;

   IF v_department_id = 60 THEN -- IT department
      IF v_salary > 9000 THEN
         DBMS_OUTPUT.PUT_LINE('Senior IT Staff');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Junior IT Staff');
      END IF;
   ELSE
      IF v_salary > 10000 THEN
         DBMS_OUTPUT.PUT_LINE('Senior Non-IT Staff');
      ELSE
         DBMS_OUTPUT.PUT_LINE('Junior Non-IT Staff');
      END IF;
   END IF;
END;
```

In this example, we first check the employee's department (like choosing between beach or mountains). Then, based on that, we check their salary to further categorize them (like choosing activities).

The Power of Nesting: Creating Complex Decision Trees

Nested IF statements allow you to create complex decision trees. Let's expand our example to include more levels:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_salary NUMBER;
   v_department_id NUMBER;
   v_job_id VARCHAR2(10);
BEGIN
   SELECT salary, department_id, job_id
   INTO v_salary, v_department_id, v_job_id
   FROM employees
   WHERE employee_id = v_employee_id;

   IF v_department_id = 60 THEN -- IT department
      IF v_salary > 9000 THEN
         IF v_job_id LIKE '%PROG%' THEN
            DBMS_OUTPUT.PUT_LINE('Senior IT Programmer');
         ELSE
            DBMS_OUTPUT.PUT_LINE('Senior IT Staff (Non-Programmer)');
         END IF;
      ELSE
         DBMS_OUTPUT.PUT_LINE('Junior IT Staff');
      END IF;
   ELSE
      IF v_salary > 10000 THEN
         IF v_job_id LIKE '%MAN%' THEN
            DBMS_OUTPUT.PUT_LINE('Senior Manager');
         ELSE
            DBMS_OUTPUT.PUT_LINE('Senior Non-Management Staff');
         END IF;
      ELSE
         DBMS_OUTPUT.PUT_LINE('Junior Non-IT Staff');
      END IF;
   END IF;
END;
```

This script is like a travel agent who first checks the department (destination), then the salary range (accommodation type), and finally the specific job role (activities).

Best Practices for Nested IF Statements

1. **Clarity is Key**: Just as a travel itinerary should be clear, make your nested IF structures easy to follow.
2. **Limit Nesting Levels**: Try not to go beyond 3-4 levels of nesting to keep your code readable.
3. **Consider Alternatives**: For very complex decisions, consider using CASE statements or creating separate procedures.

Practice Exercise

Now it's your turn to be the travel agent of PL/SQL! Write a nested IF statement that categorizes employees based on:

1. Department (IT, Sales, or Other)
2. Salary (High if > 10000, Medium if between 5000 and 10000, Low if < 5000)
3. Years of Service (New if < 5 years, Experienced if >= 5 years)

Use the hire_date column to calculate years of service. Test your code with employee_id 107.

Hint: You can calculate years of service using:

```
ROUND((SYSDATE - hire_date) / 365)
```

Remember, just like planning the perfect vacation requires considering multiple factors, mastering nested IF statements allows you to create sophisticated decision-making processes in your PL/SQL code!

Certainly! The next section in the original lesson was "Lab 4.4: Utilizing Logical Operators". I'll rewrite this section with a more didactic approach. Here's the improved version:

Lab 4.4: Utilizing Logical Operators

After this lab, you will be able to:

- Use AND, OR, and NOT operators effectively
- Combine multiple conditions in IF statements
- Create complex, precise conditions for decision-making

Imagine you're a detective solving a complex case. You need to piece together multiple clues to reach a conclusion. In PL/SQL, logical operators are your investigative tools, allowing you to combine multiple conditions to make precise decisions.

Understanding Logical Operators: The Detective's Toolkit

In PL/SQL, we have three main logical operators:

1. AND: Like finding two matching fingerprints
2. OR: Like having either a witness OR video evidence
3. NOT: Like ruling out a suspect

Let's see how these work in PL/SQL:

```
IF (condition1 AND condition2) THEN
   -- Both conditions must be true
ELSIF (condition1 OR condition2) THEN
   -- At least one condition must be true
ELSIF NOT condition THEN
   -- The condition must be false
END IF;
```

The AND Operator: Finding the Perfect Match

The AND operator is like looking for a suspect who matches all your criteria. Both conditions must be true for the overall condition to be true.

Let's use this in a practical example:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_salary NUMBER;
   v_department_id NUMBER;
BEGIN
   SELECT salary, department_id INTO v_salary, v_department_id
   FROM employees
   WHERE employee_id = v_employee_id;

   IF (v_salary > 10000 AND v_department_id = 60) THEN
      DBMS_OUTPUT.PUT_LINE('High-paid IT employee');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee does not match criteria');
   END IF;
END;
```

In this case, we're looking for employees who are both high-paid AND in the IT department.

The OR Operator: Casting a Wider Net

The OR operator is like considering multiple pieces of evidence. If any one of the conditions is true, the overall condition is true.

Let's modify our example:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_salary NUMBER;
   v_department_id NUMBER;
BEGIN
   SELECT salary, department_id INTO v_salary, v_department_id
   FROM employees
   WHERE employee_id = v_employee_id;

   IF (v_salary > 10000 OR v_department_id = 60) THEN
      DBMS_OUTPUT.PUT_LINE('Employee is either high-paid or in IT');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee is neither high-paid nor in IT');
   END IF;
END;
```

Now we're identifying employees who are either high-paid OR in the IT department.

The NOT Operator: Ruling Out Possibilities

The NOT operator is used to negate a condition. It's like saying "everyone except...".

Here's an example:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_department_id NUMBER;
BEGIN
   SELECT department_id INTO v_department_id
   FROM employees
   WHERE employee_id = v_employee_id;

   IF NOT (v_department_id = 60) THEN
      DBMS_OUTPUT.PUT_LINE('Employee is not in IT department');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee is in IT department');
   END IF;
END;
```

This identifies employees who are not in the IT department.

Combining Logical Operators: Solving the Complex Case

Just like a detective might use multiple types of evidence, we can combine these operators for more complex conditions:

```
DECLARE
   v_employee_id NUMBER := 101;
   v_salary NUMBER;
   v_department_id NUMBER;
   v_job_id VARCHAR2(10);
BEGIN
   SELECT salary, department_id, job_id INTO v_salary, v_department_id, v_job_id
   FROM employees
   WHERE employee_id = v_employee_id;

   IF (v_salary > 10000 AND v_department_id = 60) OR
     (v_job_id LIKE '%MANAGER%' AND NOT v_department_id = 30) THEN
      DBMS_OUTPUT.PUT_LINE('Employee meets complex criteria');
   ELSE
      DBMS_OUTPUT.PUT_LINE('Employee does not meet complex criteria');
   END IF;
END;
```

This complex condition identifies employees who are either high-paid IT staff OR managers outside of the Sales department.

Practice Exercise: Be the SQL Detective

Now it's your turn to be the detective! Write a PL/SQL block that identifies "key employees" based on the following criteria:

- They must be in either the IT (60) or Sales (80) department
- They must either earn more than $10,000 OR have a job title containing 'MANAGER'
- They must NOT have been hired in the last 5 years

Use employee_id 145 for this exercise.

Hint: You can check if an employee was hired in the last 5 years using:

```
hire_date > ADD_MONTHS(SYSDATE, -12*5)
```

Remember, just like a skilled detective can solve complex cases by combining different pieces of evidence, you can create sophisticated and precise conditions in your PL/SQL code by mastering logical operators!

Chapter 4 Summary: Mastering Conditional Logic in PL/SQL

Congratulations! You've journeyed through the world of conditional logic in PL/SQL. Let's recap the key concepts we've explored:

1. IF Statements: The Basic Decision Maker

We started with IF statements, the fundamental tool for decision-making in PL/SQL. Remember:

- IF-THEN for simple, one-way decisions
- IF-THEN-ELSE for two-way decisions
- Always handle NULL values carefully

Just like a bouncer at a club, IF statements allow your code to make straightforward decisions based on specific conditions.

2. ELSIF Statements: The Versatile Problem Solver

We then moved to ELSIF statements, which allow for multiple condition checking. Key points:

- Use ELSIF when you have multiple, mutually exclusive conditions
- The order of conditions matters – they're checked sequentially
- The ELSE clause catches any cases not covered by the IF or ELSIF conditions

Think of ELSIF statements as a hotel receptionist efficiently assigning rooms based on various guest preferences.

3. Nested IF Statements: The Complex Decision Tree

We delved into nested IF statements, allowing for layered decision-making. Remember:

- Nesting allows for more complex, hierarchical decisions
- Don't nest too deeply – aim for a maximum of 3-4 levels for readability
- Consider alternatives like CASE statements for very complex scenarios

Nested IF statements are like planning a detailed travel itinerary, making decisions at multiple levels.

4. Logical Operators: The Precision Tools

Finally, we explored logical operators to create precise, complex conditions:

- AND: Both conditions must be true
- OR: At least one condition must be true
- NOT: Negates a condition

These operators are your detective toolkit, allowing you to combine multiple pieces of evidence to reach a conclusion.

Putting It All Together

By mastering these concepts, you've gained the ability to write PL/SQL code that can make sophisticated decisions. You can now:

- Handle simple yes/no scenarios with basic IF statements
- Manage multiple conditions efficiently with ELSIF
- Create complex decision trees with nested IF statements
- Fine-tune your conditions with logical operators

Best Practices to Remember

1. **Clarity is Key**: Write your conditional logic so that others (including future you) can easily understand it.
2. **Test Thoroughly**: Ensure all possible scenarios are covered, including edge cases and NULL values.
3. **Optimize for Readability**: Sometimes, multiple simple IF statements are clearer than one complex nested IF.
4. **Consider Performance**: For large datasets, consider the order of your conditions to check the most likely scenarios first.

Mastering Conditional Logic in PL/SQL