# Chapter 14

## Advanced Trigger Concepts

Mayko Freitas da Silva

## Introduction

Welcome to the fascinating world of advanced database triggers. In this chapter, you'll delve deeper into the intricacies of Oracle PL/SQL triggers, focusing on two critical concepts: mutating tables and compound triggers. These advanced topics will expand your understanding of database management and help you create more robust and efficient database systems.

Imagine you're a skilled chef in a bustling kitchen. You know how to prepare dishes (like using basic triggers), but now you're about to learn how to handle complex situations during the busiest hours. That's what mutating tables and compound triggers are all about in the database world.

In this chapter, you will learn about:

1. Mutating Tables: What happens when a trigger tries to modify the very table it's watching? It's like trying to reorganize your kitchen while cooking a five-course meal.
2. Compound Triggers: These are like Swiss Army knives for your database, allowing you to combine different types of triggers into one powerful tool.

We'll explore these concepts using familiar tables from the HR schema, such as EMPLOYEES and DEPARTMENTS. By the end of this chapter, you'll be equipped to handle complex database scenarios that might have seemed daunting before.

So, roll up your sleeves and get ready to master these advanced trigger concepts. Your journey to becoming a PL/SQL trigger expert continues here.

This introduction sets the stage for the chapter, introduces the main topics, and uses analogies to make the concepts more relatable. It also mentions that we'll be using the HR schema for our examples.

## Lab 14.1: Understanding Table Mutations

After this lab, you will be able to:

- Define and identify mutating tables
- Explain the causes and nature of mutating table errors
- Recognize the limitations of row-level triggers when dealing with mutating tables

## What is a Mutating Table?

Imagine you're updating an employee's salary in the EMPLOYEES table. While this update is happening, the table is in a state of change - it's mutating. A mutating table is simply a table that is in the process of being modified by a Data Manipulation Language (DML) statement such as INSERT, UPDATE, or DELETE.

## Mutating Table Errors

Now, here's where it gets tricky. If you have a row-level trigger on the EMPLOYEES table that tries to query or modify the same table during this update, Oracle will raise a mutating table error. It's like trying to count how many employees you have while you're in the middle of hiring or firing someone - the numbers just won't add up correctly!

For example, consider this trigger on the EMPLOYEES table:

```
CREATE OR REPLACE TRIGGER check_salary_trigger
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
DECLARE
  v_avg_salary NUMBER;
BEGIN
  SELECT AVG(salary) INTO v_avg_salary
  FROM employees
  WHERE department_id = :NEW.department_id;

  IF :NEW.salary > v_avg_salary * 2 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Salary exceeds twice the department average.');
  END IF;
END;
/
```

This trigger seems logical. It's trying to ensure that no employee's salary is more than twice the average salary in their department. However, if you try to update an employee's salary, you'll encounter a mutating table error.

## Why Does This Error Occur?

## The mutating table error occurs because:

**1. The trigger fires for each row being updated (it's a row-level trigger).**

2. Inside the trigger, we're querying the EMPLOYEES table to calculate the average salary.
3. But the EMPLOYEES table is in the middle of being updated - it's mutating.

Oracle doesn't allow this because it could lead to inconsistent results. It's like trying to take a photograph of a moving object - you might catch it in an in-between state that doesn't accurately represent reality.

## Limitations of Row-Level Triggers

This example highlights a key limitation of row-level triggers: they cannot query or modify the table on which they are defined if that table is mutating. This restriction exists to prevent logical inconsistencies in your data.

It's important to note that this is a runtime error. The trigger will compile successfully, but it will raise an error when it executes during a DML operation.

## So, What Can We Do?

You might be thinking, "But I really need to perform this kind of check. How can I do it?" That's where compound triggers come in. They provide a way to work around this limitation, which we'll explore in the next lab.

Remember, understanding mutating tables and their limitations is crucial for designing efficient and error-free database triggers. In the next lab, we'll learn how to overcome these limitations using compound triggers.

This section introduces the concept of mutating tables, explains why mutating table errors occur, and highlights the limitations of row-level triggers. It uses the EMPLOYEES table from the HR schema for the example, as requested.

## Lab 14.2: Mastering Compound Triggers

After this lab, you will be able to:

- Describe the structure and purpose of compound triggers
- Implement compound triggers to resolve mutating table issues
- Understand the different timing point sections of compound triggers
- Recognize restrictions and best practices for using compound triggers

### Introduction to Compound Triggers

Remember our chef analogy? If a regular trigger is like a chef focusing on one task, a compound trigger is like a master chef orchestrating the entire kitchen. Compound triggers allow you to define multiple trigger actions for different timing points in a single trigger body. This powerful feature can help us solve mutating table issues and more.

### Structure of a Compound Trigger

A compound trigger consists of several sections, each corresponding to a different timing point. Here's the basic structure:

```
CREATE OR REPLACE TRIGGER trigger_name
FOR INSERT OR UPDATE OR DELETE ON table_name
COMPOUND TRIGGER

  -- Declaration Section
  -- Variables declared here are accessible in all sections

  BEFORE STATEMENT IS
  BEGIN
    -- Executes once before the triggering statement
  END BEFORE STATEMENT;

  BEFORE EACH ROW IS
  BEGIN
    -- Executes before each row affected by the triggering statement
  END BEFORE EACH ROW;

  AFTER EACH ROW IS
  BEGIN
    -- Executes after each row affected by the triggering statement
  END AFTER EACH ROW;

  AFTER STATEMENT IS
  BEGIN
    -- Executes once after the triggering statement
  END AFTER STATEMENT;

END;
/
```

## Solving the Mutating Table Problem

Let's revisit our salary check problem from the previous lab, but this time using a compound trigger:

```
CREATE OR REPLACE TRIGGER check_salary_compound_trigger
FOR INSERT OR UPDATE ON employees
COMPOUND TRIGGER

  -- Declaration
  TYPE t_salary_changes IS TABLE OF employees.salary%TYPE INDEX BY PLS_INTEGER;
  l_salary_changes t_salary_changes;
  l_row_count PLS_INTEGER := 0;

  -- Before Each Row
  BEFORE EACH ROW IS
  BEGIN
    l_row_count := l_row_count + 1;
    l_salary_changes(l_row_count) := :NEW.salary;
  END BEFORE EACH ROW;

  -- After Statement
  AFTER STATEMENT IS
    l_avg_salary NUMBER;
    l_max_salary NUMBER;
  BEGIN
    SELECT AVG(salary), MAX(salary) INTO l_avg_salary, l_max_salary
    FROM employees;

    FOR i IN 1..l_row_count LOOP
      IF l_salary_changes(i) > l_avg_salary * 2 THEN
        RAISE_APPLICATION_ERROR(-20001, 'An employee salary exceeds twice the average.');
      END IF;
    END LOOP;
  END AFTER STATEMENT;

END;
/
```

## This compound trigger solves our mutating table problem by:

**1. Collecting the new salaries in the BEFORE EACH ROW section.**

2. Performing the average salary check in the AFTER STATEMENT section, when the table is no longer mutating.

## Key Points About Compound Triggers

1. **Shared Variables**: Variables declared in the declaration section are accessible in all timing-point sections.
2. **Pseudorecords**: :NEW and :OLD are only available in the row-level sections (BEFORE EACH ROW and AFTER EACH ROW).
3. **Exception Handling**: Each section must handle its own exceptions. An exception raised in one section doesn't propagate to other sections.
4. **Restrictions**:
    - Compound triggers can only be defined on tables or views.
    - They can't contain autonomous transactions.
    - The firing order between compound and simple triggers is not guaranteed.
5. **Rollback Behavior**: If a DML statement fails and rolls back:
    - Variables in the compound trigger are reinitialized.
    - DML statements issued by the compound trigger are not rolled back automatically.

## Best Practices

1. Use compound triggers to solve mutating table issues.
2. Keep the logic in each section focused and simple.
3. Be cautious with DML operations in compound triggers, as they won't automatically roll back if the triggering statement fails.
4. Always test thoroughly, especially when dealing with large datasets.

Compound triggers are powerful tools in your PL/SQL toolkit. They allow you to overcome limitations of simple triggers and provide more control over the timing and execution of your trigger logic. With great power comes great responsibility - use them wisely!

This section introduces compound triggers, explains their structure, demonstrates how they can solve mutating table problems, and covers key points and best practices. The example uses the EMPLOYEES table from the HR schema, as requested.

## Summary

In this chapter, you've explored two advanced concepts in Oracle PL/SQL triggers: mutating tables and compound triggers. Let's recap the key points:

1. **Mutating Tables**:
    - A mutating table is a table that's currently being modified by a DML operation.
    - Row-level triggers can cause mutating table errors when they try to query or modify the table on which they're defined.
    - These errors occur at runtime, not during compilation.
    - Mutating table errors prevent logical inconsistencies but can limit trigger functionality.
2. **Compound Triggers**:
    - Compound triggers allow you to define multiple trigger actions for different timing points in a single trigger body.
    - They have four main sections: BEFORE STATEMENT, BEFORE EACH ROW, AFTER EACH ROW, and AFTER STATEMENT.
    - Compound triggers can solve mutating table issues by deferring operations to the statement-level sections.
    - They allow shared variables across different timing point sections.
    - Compound triggers have specific restrictions and behaviors, especially regarding exception handling and rollbacks.
3. **Practical Application**:
    - You learned how to transform a problematic row-level trigger into a compound trigger to avoid mutating table errors.
    - The example of salary checking in the EMPLOYEES table demonstrated how to collect data at the row level and process it at the statement level.
4. **Best Practices**:
    - Use compound triggers to overcome limitations of simple triggers, especially for mutating table scenarios.
    - Keep logic in each section of a compound trigger focused and simple.
    - Be cautious with DML operations in compound triggers due to their rollback behavior.
    - Always thoroughly test triggers, especially with large datasets.

By mastering these concepts, you've significantly expanded your ability to create sophisticated database triggers. You can now handle complex scenarios that require cross-row or cross-statement logic, all while maintaining data consistency and integrity.

Remember, triggers are powerful tools in database programming, but they should be used judiciously. Always consider the performance implications and potential side effects when implementing complex trigger logic.

As you continue to work with the HR schema and other databases, these advanced trigger concepts will allow you to create more robust, efficient, and error-free database systems. Keep practicing and exploring these concepts to become a true PL/SQL trigger expert!

This summary recaps the main points covered in the chapter, emphasizing the key concepts of mutating tables and compound triggers. It also reinforces the practical application using the HR schema and reiterates best practices.

## Advanced Trigger Concepts