Chapter 13

Database Triggers: Your Automatic Assistants

Mayko Freitas da Silva

Introduction

Imagine you're setting up a smart home system. You program it so that when you open the front door, the lights automatically turn on, the thermostat adjusts to your preferred temperature, and your favorite music starts playing. This automatic response to an event is similar to how database triggers work in the world of data management.

In this chapter, you'll dive into the fascinating world of database triggers. These powerful tools act as your automatic assistants, quietly working behind the scenes to maintain data integrity, enforce business rules, and automate repetitive tasks in your database.

You might be wondering, "Why do I need triggers when I can write code to do these things manually?" Well, triggers offer several advantages:

1. Consistency: They ensure that certain actions always occur when specific events happen, regardless of how the data is modified.
2. Efficiency: Triggers automate processes, reducing the need for manual intervention and minimizing human error.
3. Centralization: By placing logic in triggers, you can enforce rules across all applications that interact with the database.

Throughout this chapter, you'll learn how to harness the power of triggers to make your database smarter and more responsive. You'll discover different types of triggers, when to use each one, and how to create them effectively.

In this chapter, you will learn about:

- What triggers are and how they work
- The different types of triggers: BEFORE, AFTER, and INSTEAD OF
- How to create and implement triggers
- Advanced concepts like autonomous transactions
- The difference between row-level and statement-level triggers
- How to use triggers with views

By the end of this chapter, you'll have the knowledge and skills to create your own database assistants, ready to spring into action whenever you need them. So, let's get started on this exciting journey into the world of database triggers.

Lab 13.1: Getting to Know Your Triggers

After this lab, you will be able to:

- Define what a database trigger is
- Identify different types of triggering events
- Understand the basic syntax for creating triggers
- Create and implement simple BEFORE and AFTER triggers

What Are Triggers?

Imagine you have a faithful assistant who's always watching your database, ready to take action whenever something important happens. That's essentially what a database trigger is - a special kind of stored procedure that automatically executes when a specific event occurs in the database.

A database trigger is like a vigilant guardian for your data. It's a named PL/SQL block that's stored in the database and executed (or "fired") implicitly when a triggering event occurs. But what exactly can cause a trigger to fire? Let's look at some common triggering events:

1. DML operations: These are your everyday data modifications.
    - INSERT: When new data is added to a table
    - UPDATE: When existing data in a table is modified
    - DELETE: When data is removed from a table
2. DDL operations: These involve changes to the database structure.
    - CREATE: When a new database object is created
    - ALTER: When an existing database object is modified
    - DROP: When a database object is removed
3. System events:
    - Database startup
    - Database shutdown
4. User events:
    - User login
    - User logoff

Now, let's look at the general syntax for creating a trigger:

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} triggering_event
ON table_name
[FOR EACH ROW]
[WHEN condition]
BEGIN
  -- Trigger body
END;
```

Let's break this down:

- CREATE [OR REPLACE] TRIGGER: This is how you start creating a new trigger (or replacing an existing one).
- trigger_name: This is the name you give your trigger.
- {BEFORE | AFTER | INSTEAD OF}: This specifies when the trigger should fire relative to the triggering event.
- triggering_event: This is the event that causes the trigger to fire (like INSERT, UPDATE, DELETE).
- ON table_name: This specifies which table the trigger is associated with.
- [FOR EACH ROW]: If included, this makes the trigger a row-level trigger (more on this later).
- [WHEN condition]: This is an optional condition that must be true for the trigger to fire.
- BEGIN ... END: This is where you put the PL/SQL code that the trigger will execute.

Now that you understand the basics, let's create your first trigger!

Creating Your First Trigger

Let's create a simple trigger on the EMPLOYEES table in the HR schema. This trigger will automatically set the HIRE_DATE to the current date whenever a new employee is inserted.

```
CREATE OR REPLACE TRIGGER trg_set_hire_date
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
  :NEW.hire_date := SYSDATE;
END;
/
```

Let's break down what this trigger does:

1. It's named trg_set_hire_date.
2. It fires BEFORE an INSERT operation on the EMPLOYEES table.
3. It's a row-level trigger (FOR EACH ROW), meaning it fires once for each row affected by the INSERT.
4. In the trigger body, it sets the HIRE_DATE of the new row (:NEW.hire_date) to the current date (SYSDATE).

Now, whenever you insert a new employee without specifying a HIRE_DATE, this trigger will automatically set it to the current date. This ensures consistency and saves you from having to manually enter the date each time.

BEFORE Triggers: Setting the Stage

BEFORE triggers are like the stagehands in a theater, working behind the scenes to ensure everything is in order before the main action takes place. They fire before the triggering SQL statement is executed, allowing you to modify the data or perform checks before changes are made to the database.

Key uses of BEFORE triggers include:

1. Automatically generating values for columns

2. Enforcing complex business rules
3. Validating or modifying input data before it's stored

Let's create a more complex BEFORE trigger on the EMPLOYEES table. This trigger will ensure that an employee's salary is within an acceptable range based on their job before an INSERT or UPDATE operation.

```
CREATE OR REPLACE TRIGGER trg_validate_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
DECLARE
  v_min_salary employees.salary%TYPE;
  v_max_salary employees.salary%TYPE;
BEGIN
  -- Get the salary range for the employee's job
  SELECT min_salary, max_salary
  INTO v_min_salary, v_max_salary
  FROM jobs
  WHERE job_id = :NEW.job_id;

  -- Check if the new salary is within the acceptable range
  IF :NEW.salary < v_min_salary OR :NEW.salary > v_max_salary THEN
    RAISE_APPLICATION_ERROR(-20001, 'Salary $' || :NEW.salary ||
      ' is out of range for job ' || :NEW.job_id ||
      ' ($' || v_min_salary || ' - $' || v_max_salary || ')');
  END IF;
END;
/
```

This trigger does the following:

1. It fires before an INSERT or an UPDATE of the salary column.

2. It retrieves the minimum and maximum salary for the employee's job from the JOBS table.
3. It checks if the new salary is within this range.
4. If the salary is out of range, it raises an error with a descriptive message.

Now, let's try to insert a new employee with a salary that's too high:

```
INSERT INTO employees (employee_id, first_name, last_name, email, phone_number,
                      hire_date, job_id, salary, commission_pct, manager_id, department_id)
VALUES (employees_seq.NEXTVAL, 'John', 'Doe', 'JDOE', '515.123.4567',
       SYSDATE, 'IT_PROG', 30000, NULL, 103, 60);
```

This should raise an error because the salary is too high for an IT_PROG job.

AFTER Triggers: Cleaning Up

AFTER triggers are like the cleanup crew that comes in after an event. They fire after the triggering SQL statement has been executed, allowing you to perform actions based on the changes that were just made to the database.

Key uses of AFTER triggers include:

1. Auditing changes to data

2. Maintaining summary tables
3. Triggering cascading changes in related tables

Let's create an AFTER trigger that maintains an audit log of salary changes:

```
CREATE TABLE salary_changes_log (
  log_id        NUMBER PRIMARY KEY,
  employee_id   NUMBER,
  old_salary    NUMBER,
  new_salary    NUMBER,
  change_date   DATE
);

CREATE SEQUENCE salary_changes_log_seq;

CREATE OR REPLACE TRIGGER trg_log_salary_changes
AFTER UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  INSERT INTO salary_changes_log (log_id, employee_id, old_salary, new_salary, change_date)
  VALUES (salary_changes_log_seq.NEXTVAL, :OLD.employee_id, :OLD.salary, :NEW.salary, SYSDATE);
END;
/
```

This trigger does the following:

1. It fires after an UPDATE of the salary column in the EMPLOYEES table.

2. For each updated row, it inserts a new record into the SALARY_CHANGES_LOG table.
3. It uses the :OLD and :NEW pseudorecords to access the old and new salary values.

Now, let's update an employee's salary and see the log entry:

```
UPDATE employees SET salary = salary * 1.1 WHERE employee_id = 103;

SELECT * FROM salary_changes_log WHERE employee_id = 103;
```

This should show you the log entry for the salary change.

Exercise

Now it's your turn to practice. Create a BEFORE trigger on the DEPARTMENTS table that automatically sets the MANAGER_ID to NULL if the specified MANAGER_ID doesn't exist in the EMPLOYEES table. Then, create an AFTER trigger that logs all changes (inserts, updates, and deletes) to the DEPARTMENTS table in a new audit table.

In the next lab, we'll explore more advanced trigger concepts, including autonomous transactions and INSTEAD OF triggers.

Lab 13.2: Advanced Trigger Techniques

After this lab, you will be able to:

- Implement autonomous transactions in triggers
- Distinguish between row-level and statement-level triggers
- Create INSTEAD OF triggers for views

Autonomous Transactions: Independent Actions

Imagine you're a bank teller processing a large withdrawal. While you're counting out the cash, you need to quickly jot down the transaction in a separate ledger, regardless of whether the main transaction is completed or cancelled. This is similar to how autonomous transactions work in triggers.

An autonomous transaction is an independent transaction started by another transaction, called the main transaction. It allows a trigger to perform database operations that are committed or rolled back independently of the main transaction.

Here's how you can create an autonomous transaction in a trigger:

```
CREATE TABLE emp_audit_log (
    log_id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    action_date DATE,
    action VARCHAR2(10),
    employee_id NUMBER,
    performed_by VARCHAR2(30)
);

CREATE OR REPLACE TRIGGER trg_log_emp_changes
AFTER INSERT OR UPDATE OR DELETE ON employees
FOR EACH ROW
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
    v_action VARCHAR2(10);
BEGIN
    -- Determine the action (INSERT, UPDATE, or DELETE)
    IF INSERTING THEN
        v_action := 'INSERT';
    ELSIF UPDATING THEN
        v_action := 'UPDATE';
    ELSIF DELETING THEN
        v_action := 'DELETE';
    END IF;

    -- Log the action
    INSERT INTO emp_audit_log (action_date, action, employee_id, performed_by)
    VALUES (SYSDATE, v_action, :NEW.employee_id, USER);

    -- Commit the autonomous transaction
    COMMIT;
END;
```

Row vs. Statement Triggers: Granular Control

Row-level and statement-level triggers give you different levels of control over when and how often your trigger fires.

1. Row-level triggers:
    - Fire once for each row affected by the triggering statement
    - Use the FOR EACH ROW clause
    - Can use the :OLD and :NEW pseudorecords
2. Statement-level triggers:
    - Fire once per triggering statement, regardless of how many rows are affected
    - Do not use the FOR EACH ROW clause
    - Cannot use :OLD and :NEW pseudorecords

Let's create a statement-level trigger that logs the number of rows affected by bulk operations on the EMPLOYEES table:

```
CREATE OR REPLACE TRIGGER trg_log_bulk_emp_changes
AFTER INSERT OR UPDATE OR DELETE ON employees
DECLARE
  v_action VARCHAR2(10);
  v_rows_affected NUMBER;
BEGIN
  -- Determine the action and number of rows affected
  IF INSERTING THEN
    v_action := 'INSERT';
    v_rows_affected := SQL%ROWCOUNT;
  ELSIF UPDATING THEN
    v_action := 'UPDATE';
    v_rows_affected := SQL%ROWCOUNT;
  ELSIF DELETING THEN
    v_action := 'DELETE';
    v_rows_affected := SQL%ROWCOUNT;
  END IF;

  -- Log the bulk action
  INSERT INTO emp_bulk_audit_log (action_date, action, rows_affected, performed_by)
  VALUES (SYSDATE, v_action, v_rows_affected, USER);
END;
/
```

This trigger fires once per statement and logs the total number of rows affected by bulk INSERT, UPDATE, or DELETE operations.

Pro Tips:

1. Row-Level Triggers

- **Purpose**: Ideal for tasks requiring specific actions on **each row** affected by a DML operation (INSERT, UPDATE, DELETE).
- **Examples**:
  - Validating data against complex rules before it is committed.
  - Automatically calculating derived or dependent fields in other tables.

⚠️ **Caution**: When used with bulk operations (e.g., large-scale UPDATE or DELETE), row-level triggers can degrade performance due to the overhead of executing the trigger for every row.

2. Statement-Level Triggers

- **Purpose**: Best for tasks that operate at a higher level, involving the entire DML statement rather than individual rows.
- **Examples**:
  - Auditing who performed a bulk operation.
  - Logging when specific tables were modified without needing to know which rows were affected.

⚠️ **Caution**: Statement-level triggers do not have access to :NEW or :OLD pseudo-records because they don't deal with individual row data.

3. Performance Considerations with Row-Level Triggers in Bulk Operations

- Bulk operations like INSERT ALL, large UPDATE, or MERGE statements can trigger row-level logic thousands (or millions) of times.
- **Solution**: Where possible, consolidate logic into a **single SQL statement** or use **statement-level triggers** for better scalability.

4. Using SQL%ROWCOUNT in BEFORE Statement-Level Triggers

- In **BEFORE** triggers, the rows haven't been processed yet, so SQL%ROWCOUNT is always 0.
- **Alternative**: Use an **AFTER statement-level trigger** if you need to access SQL%ROWCOUNT to determine the number of rows affected.

INSTEAD OF Triggers: Alternate Routes for Views

INSTEAD OF triggers are special triggers that you can create on views. They allow you to define custom logic that executes instead of the default INSERT, UPDATE, or DELETE operations on the view.

Let's create a view and an INSTEAD OF trigger to make it updatable:

```
-- Create a view joining EMPLOYEES and DEPARTMENTS{

CREATE OR REPLACE VIEW emp_dept_view AS
SELECT e.employee_id, e.first_name, e.last_name, e.salary,
       d.department_id, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;

-- Create an INSTEAD OF trigger to handle updates to the view
CREATE OR REPLACE TRIGGER trg_update_emp_dept_view
INSTEAD OF UPDATE ON emp_dept_view
FOR EACH ROW
BEGIN
   -- Update the EMPLOYEES table
   UPDATE employees
   SET salary = :NEW.salary
   WHERE employee_id = :OLD.employee_id;

   -- Update the DEPARTMENTS table
   UPDATE departments
   SET department_name = :NEW.department_name
   WHERE department_id = :OLD.department_id;
END;
/
```

This trigger allows you to update both the EMPLOYEES and DEPARTMENTS tables through the EMP_DEPT_VIEW view. When you update the view, the trigger intercepts the operation and performs the appropriate updates on the underlying tables.

Now you can update the view like this:

```
UPDATE emp_dept_view
SET salary = 5000, department_name = 'New IT'
WHERE employee_id = 103;
```

This will update both the salary in the EMPLOYEES table and the department name in the DEPARTMENTS table.

Exercise

Create a complex INSTEAD OF trigger for a view that joins the EMPLOYEES, DEPARTMENTS, and LOCATIONS tables. The trigger should handle INSERT, UPDATE, and DELETE operations on the view, updating the underlying tables appropriately. Consider potential constraints and error handling in your trigger.

Summary

In this chapter, you've embarked on an exciting journey into the world of database triggers. Let's recap the key concepts you've learned:

1. **What are Triggers?**
    - Triggers are special stored procedures that automatically execute when specific events occur in the database.
    - They act as automatic assistants, helping maintain data integrity, enforce business rules, and automate tasks.
2. **Types of Triggers**
    - BEFORE triggers: Execute before the triggering event, useful for data validation and modification.
    - AFTER triggers: Execute after the triggering event, ideal for auditing and maintaining derived data.
    - INSTEAD OF triggers: Used with views to define custom logic for INSERT, UPDATE, and DELETE operations.
3. **Trigger Syntax**

    You learned the basic syntax for creating triggers and how to specify when they should fire.

4. **BEFORE and AFTER Triggers**
    - You created BEFORE triggers to validate data and set default values.
    - You implemented AFTER triggers for auditing and maintaining summary data.
5. **Autonomous Transactions**

    You learned how to use autonomous transactions within triggers to perform independent database operations.

6. **Row vs. Statement Triggers**
    - Row-level triggers: Fire once for each affected row, can use :OLD and :NEW pseudorecords.
    - Statement-level triggers: Fire once per triggering statement, useful for logging bulk operations.

7. **INSTEAD OF Triggers**

    You created INSTEAD OF triggers to make complex views updatable by defining custom logic for DML operations.

Throughout these labs, you've gained practical experience by:

- Creating triggers to automatically set hire dates and validate salaries.
- Implementing audit logs for salary changes and bulk operations.
- Making views updatable using INSTEAD OF triggers.

Remember, while triggers are powerful tools, they should be used judiciously. Overuse of triggers can lead to complex, hard-to-maintain database systems. Always consider whether a trigger is the best solution for your specific use case.

As you continue your journey in database development, you'll find triggers to be invaluable tools for maintaining data integrity, enforcing business rules, and creating more responsive database systems. Practice creating and using triggers in various scenarios to solidify your understanding and skills.

Database Triggers: Your Automatic Assistants