Study Guide: Triggers in SQL

Prepared by Andrés Calderón

May 19, 2025

1 Introduction to Triggers

A **trigger** is a database object that automatically executes a predefined action in response to certain events on a table or view. It is a type of stored procedure that is triggered automatically when a specified database operation occurs.

Key Features

- Automated execution
- Event-driven
- Typically used for enforcing business rules, maintaining audit trails, and validating data

2 Why Use Triggers?

Triggers are particularly useful for:

- **Data Integrity:** Enforcing complex constraints that cannot be captured by primary keys, unique constraints, or check constraints.
- **Audit Trails:** Automatically recording changes to critical data.
- **Business Rules:** Implementing complex logic at the database level.
- **Synchronization:** Keeping related tables consistent.
- **Security:** Validating data before changes are made.

3 Types of Triggers

Triggers can be classified based on the timing and event that triggers their execution:

Timing

- **BEFORE Trigger:** Executes before the triggering event.
- **AFTER Trigger:** Executes after the triggering event.
- **INSTEAD OF Trigger:** Replaces the triggering event (typically for views).

Event

- **INSERT Trigger:** Fires when a new row is inserted.
- **UPDATE Trigger:** Fires when an existing row is updated.
- **DELETE Trigger:** Fires when a row is deleted.

Nested Triggers

A **nested trigger** is a trigger that is fired as a result of another trigger. This occurs when the execution of one trigger causes a data modification (INSERT, UPDATE, or DELETE) that activates a second trigger on the same or a different table.

Key Characteristics

- **Chained Execution:** Triggers can form a chain where one trigger's action initiates another.
- **Depth Control:** Some database systems, like SQL Server, allow you to limit the nesting level to prevent infinite loops.
- **Performance Impact:** Excessive nesting can lead to performance issues and complex debugging.
- **Transactional Context:** Nested triggers operate within the same transaction, meaning a failure in a nested trigger can roll back the entire chain.

Example: Nested Trigger in PostgreSQL
This example demonstrates how a trigger on one table can cause a second trigger on a related table to activate.

```sql
-- Table for tracking customer status changes
CREATE TABLE customer_status_changes (
    change_id SERIAL PRIMARY KEY,
    customer_id INT,
    old_status VARCHAR(50),
    new_status VARCHAR(50),
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Table for logging audit history
CREATE TABLE audit_log (
    log_id SERIAL PRIMARY KEY,
    message TEXT,
    logged_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
16
17  -- Trigger function to log status changes
18  CREATE OR REPLACE FUNCTION log_status_changes()
19  RETURNS TRIGGER AS $$
20  BEGIN
21      INSERT INTO customer_status_changes (customer_id, old_status, new_status)
22      VALUES (OLD.customer_id, OLD.status, NEW.status);
23      RETURN NEW;
24  END;
25  $$ LANGUAGE plpgsql;
26
27  -- Trigger function for nested logging
28  CREATE OR REPLACE FUNCTION log_audit_message()
29  RETURNS TRIGGER AS $$
30  BEGIN
31      INSERT INTO audit_log (message)
32      VALUES ('Customer status updated for customer ID: ' || NEW.customer_id);
33      RETURN NEW;
34  END;
35  $$ LANGUAGE plpgsql;
36
37  -- Primary trigger on customer status updates
38  CREATE TRIGGER customer_status_trigger
39  AFTER UPDATE ON customers
40  FOR EACH ROW
41  EXECUTE FUNCTION log_status_changes();
42
43  -- Nested trigger on customer status change table
44  CREATE TRIGGER nested_audit_trigger
45  AFTER INSERT ON customer_status_changes
46  FOR EACH ROW
47  EXECUTE FUNCTION log_audit_message();
```

4 Basic Syntax of Triggers (PostgreSQL)

```
1  CREATE TRIGGER trigger_name
2  { BEFORE | AFTER | INSTEAD OF } { INSERT | UPDATE | DELETE }
3  ON table_name
4  FOR EACH { ROW | STATEMENT }
5  WHEN (condition)
6  EXECUTE FUNCTION function_name();
```

5 Practical Examples

Audit Log Trigger

Tracking changes in an `employees` table:

```sql
CREATE TABLE employees_audit (
    audit_id SERIAL PRIMARY KEY,
    employee_id INT,
    operation VARCHAR(10),
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION log_employee_changes()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO employees_audit (employee_id, operation)
    VALUES (NEW.employee_id, TG_OP);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER employee_audit_trigger
AFTER INSERT OR UPDATE OR DELETE
ON employees
FOR EACH ROW
EXECUTE FUNCTION log_employee_changes();
```

Data Validation Trigger

Preventing negative salaries in the `employees` table:

```sql
CREATE OR REPLACE FUNCTION validate_salary()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION 'Salary cannot be negative';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER salary_check
BEFORE INSERT OR UPDATE
ON employees
FOR EACH ROW
EXECUTE FUNCTION validate_salary();
```

6 Transition Tables in Triggers

A **transition table** in PostgreSQL is a special table used in triggers to capture the full set of affected rows during bulk operations. Unlike the `OLD` and `NEW` row-level references, transition tables provide a set-based view of all modified rows, making them ideal for performance tuning and bulk data processing. They are available for `AFTER` triggers on `UPDATE` and `DELETE` operations.

- Useful for auditing large changes without per-row overhead.
- Enables set-based processing within triggers for better efficiency.
- Can be referenced in the trigger function using the `REFERENCING` clause.

Example: Using Transition Tables for Bulk Auditing

This example demonstrates how to use transition tables to log bulk updates to an `employees` table:

```
CREATE TABLE employees_bulk_audit (
    audit_id SERIAL PRIMARY KEY,
    employee_id INT,
    old_salary NUMERIC,
    new_salary NUMERIC,
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION log_bulk_salary_changes()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO employees_bulk_audit (employee_id, old_salary, new_salary)
    SELECT OLD.employee_id, OLD.salary, NEW.salary
    FROM OLD_TABLE AS OLD, NEW_TABLE AS NEW
    WHERE OLD.employee_id = NEW.employee_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER bulk_salary_audit
AFTER UPDATE ON employees
REFERENCING OLD TABLE AS OLD_TABLE NEW TABLE AS NEW_TABLE
FOR EACH STATEMENT
EXECUTE FUNCTION log_bulk_salary_changes();
```

7 Best Practices for Using Triggers

- Use triggers sparingly to avoid performance issues.
- Document the purpose of each trigger clearly.
- Avoid complex logic that can make debugging difficult.
- Use triggers for auditing and validation, but not as a primary business logic layer.
- Test triggers extensively before deploying to production.
- Use transition tables for bulk operations to reduce overhead.
- Monitor trigger performance regularly to avoid bottlenecks.

8 Additional Content

- *"SQL Triggers: A Beginner's Guide"*. Oluseye Jeremiah. 2024. Link in DataCamp. [3].
- *"Triggers In SQL— Triggers In Database, SQL Triggers Tutorial For Beginners"*. Edureka!, 2023. Link in YouTube. [1].
- *"¿Qué diablos es un Trigger? Ejemplo sencillo en Sql Server"*. Héctor de León. 2018. In Spanish. Link in YouTube. [2].

9 Exercises

1. Create a trigger to prevent the deletion of VIP customers.
2. Implement a trigger that logs every update to a `products` table, including the old and new price.
3. Write a trigger that automatically updates a stock table whenever a sale is made.
4. Create a trigger that validates email format before inserting a customer record.
5. Optimize a trigger for bulk data updates using transition tables.
6. Design a trigger that automatically archives deleted rows into a history table.
7. Implement a performance monitoring trigger to track long-running updates.

Solutions

Exercise 1: Preventing the Deletion of VIP Customers

To prevent the deletion of VIP customers, you can create a trigger that raises an exception if an attempt is made to delete a customer marked as VIP.

```
CREATE OR REPLACE FUNCTION prevent_vip_deletion()
RETURNS TRIGGER AS $$
BEGIN
    IF OLD.is_vip THEN
        RAISE EXCEPTION 'Cannot delete VIP customers.';
    END IF;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER vip_deletion_blocker
BEFORE DELETE ON customers
FOR EACH ROW
EXECUTE FUNCTION prevent_vip_deletion();
```

Exercise 2: Logging Product Price Updates

To log every price update in a `products` table, you can use a trigger that captures both the old and new prices.

```
CREATE TABLE product_price_log (
    log_id SERIAL PRIMARY KEY,
    product_id INT,
    old_price NUMERIC,
    new_price NUMERIC,
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION log_price_changes()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO product_price_log (product_id, old_price, new_price)
    VALUES (OLD.product_id, OLD.price, NEW.price);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER price_update_logger
AFTER UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION log_price_changes();
```

Exercise 3: Stock Update on Sale

To automatically update the stock of a product when a sale is made, you can use a trigger to adjust the inventory.

```
CREATE OR REPLACE FUNCTION update_stock_on_sale()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE products
    SET stock = stock - NEW.quantity
    WHERE product_id = NEW.product_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER stock_update_trigger
AFTER INSERT ON sales
FOR EACH ROW

EXECUTE FUNCTION update_stock_on_sale();
```

Exercise 4: Email Format Validation

To validate email formats before inserting customer records, you can create a trigger to enforce proper formatting.

```
CREATE OR REPLACE FUNCTION validate_email_format()
RETURNS TRIGGER AS $$
BEGIN
    IF NOT NEW.email ~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$' THEN
        RAISE EXCEPTION 'Invalid email format: %', NEW.email;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER email_validation_trigger
BEFORE INSERT OR UPDATE ON customers
FOR EACH ROW
EXECUTE FUNCTION validate_email_format();
```

Exercise 5: Bulk Data Updates Using Transition Tables

To efficiently handle bulk data updates, you can use transition tables to capture the old and new state of rows during a single operation.

```
CREATE TABLE bulk_update_audit (
    audit_id SERIAL PRIMARY KEY,
    product_id INT,
    old_price NUMERIC,
```

```
    new_price NUMERIC,
    change_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION log_bulk_price_changes()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO bulk_update_audit (product_id, old_price, new_price)
    SELECT OLD.product_id, OLD.price, NEW.price
    FROM OLD_TABLE AS OLD
    JOIN NEW_TABLE AS NEW
    ON OLD.product_id = NEW.product_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER bulk_price_update_trigger
AFTER UPDATE ON products
REFERENCING OLD TABLE AS OLD_TABLE NEW TABLE AS NEW_TABLE
FOR EACH STATEMENT
EXECUTE FUNCTION log_bulk_price_changes();
```

Exercise 6: Archiving Deleted Rows

To automatically archive deleted rows, you can create a trigger that stores deleted data in a history table.

```
CREATE TABLE customer_archive (
    archive_id SERIAL PRIMARY KEY,
    customer_id INT,
    name VARCHAR(255),
    email VARCHAR(255),
    deleted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION archive_deleted_customers()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO customer_archive (customer_id, name, email)
    VALUES (OLD.customer_id, OLD.name, OLD.email);
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER archive_on_delete
BEFORE DELETE ON customers
FOR EACH ROW
EXECUTE FUNCTION archive_deleted_customers();
```

Exercise 7: Performance Monitoring

To track long-running updates, you can create a trigger to log slow transactions for performance analysis.

```sql
CREATE TABLE slow_query_log (
    log_id SERIAL PRIMARY KEY,
    table_name VARCHAR(255),
    operation VARCHAR(10),
    duration_ms INT,
    logged_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION log_slow_updates()
RETURNS TRIGGER AS $$
DECLARE
    start_time TIMESTAMP;
    end_time TIMESTAMP;
    duration INT;
BEGIN
    start_time := clock_timestamp();
    -- Simulate some processing delay for demonstration
    PERFORM pg_sleep(0.5);
    end_time := clock_timestamp();
    duration := EXTRACT(MILLISECOND FROM end_time - start_time);

    IF duration > 500 THEN
        INSERT INTO slow_query_log (table_name, operation, duration_ms)
        VALUES (TG_TABLE_NAME, TG_OP, duration);
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER slow_update_logger
AFTER UPDATE OR DELETE ON customers
FOR EACH ROW
EXECUTE FUNCTION log_slow_updates();
```

References

[1] Edureka! Triggers in sql — triggers in database — sql triggers tutorial for beginners, 2023. Accessed: 2025-05-19.

[2] hdeleon.net. ¿qué diablos es un trigger? — ejemplo sencillo en sql server, 2018. Accessed: 2025-05-19.

[3] Oluseye Jeremiah. Sql triggers: A beginner's guide, 2024. Accessed: 2025-05-19.