

REPUBLIC OF CAMEROON

Peace-Work-Fatherland

Inter State Establishment of Higher Education

IAI PAUL BIYA TECHNOLOGICAL CENTER OF EXCELLENCE

Bo: 13719 Yaoundé (Cameroun)

Phone: +(237)242 72 99 57

Website: [www.iaicameroun.com](http://www.iaicameroun.com)

Email: contact@iaicameroun.com

REPUBLIC DU CAMEROUN

Paix-Travail-Patrie

Etablissements Inter Etats d’Enseignements Supérieurs

IAI CENTRE D’EXCELLENCE TECHNOLOGIQUE PAUL BIYA

BP : 13719 Yaoundé (Cameroun)

Tel : +(237)242 72 99 57

Site Web: [www.iaicameroun.com](http://www.iaicameroun.com)

Email: contact@iaicameroun.com

**REPORT**

**POSTGRESQL EXPOSE:**

**THEME: TRIGGERS IN POSTGRESQL**

Under the supervision of

**Mr Messio Ulrich**

**GROUP MEMBERS :**

* **TANGOMO CHRIST (Leader)**
* **NGOUANE JESSICA**
* **FOUDA BERTINE**
* **LOWE RWANE**
* **JEAN JONATHAN**
* **NJOULI REINE**

**GROUP MEMBERS :**

* **TANGOMO CHRIST (Leader)**
* **NGOUANE JESSICA**
* **FOUDA BERTINE**
* **LOWE RWANE**
* **JEAN JONATHAN**
* **NJOULI REINE**

**ACADEMIC YEAR**

**2024-2025**

**ACADEMIC YEAR**

**2024-2025**

**SCHEME OF WORK**

[**I.** **INTRODUCTION TO TRIGGERS IN POSTGRESQL** 2](#_Toc195644403)

[**BRIEF OVERVIEW OF TRIGGERS AND THEIR ORIGINS** 2](#_Toc195644404)

[**Origins of triggers** 2](#_Toc195644405)

[**What are triggers?** 3](#_Toc195644406)

[**Key Characteristics of Triggers** 3](#_Toc195644407)

[**Why Do we Use Triggers?** 3](#_Toc195644408)

[**II.** **TYPES OF TRIGGERS** 4](#_Toc195644409)

[**A.** **DML TRIGGERS** 4](#_Toc195644410)

[**B.** **DDL TRIGGERS** 4](#_Toc195644411)

[**C.** **EVENT TRIGGERS** 4](#_Toc195644412)

[**D.** **INSTEAD OF TRIGGERS** 4](#_Toc195644413)

[**III.** **TRIGGER FUNCTIONS** 4](#_Toc195644414)

[**A.** **REQUIREMENTS OF TRIGGER FUNCTIONS IN POSTGRESQL** 4](#_Toc195644415)

[**B.** **STRUCTURE OF TRIGGER FUNCTIONS IN POSTGRESQL** 5](#_Toc195644416)

[**C.** **CODE SNIPPETS** 6](#_Toc195644417)

[**IV.** **USE CASES AND ILLUSTRATIONS** 7](#_Toc195644418)

[**1.** **EXAMPLE: USE CASE OF AUDIT LOGGING:** 7](#_Toc195644419)

[ **What Problem Does This Solve?** 7](#_Toc195644420)

[**V.** **BEST PRACTICES** 9](#_Toc195644421)

[**A.** **WHEN TO USE TRIGGERS** 9](#_Toc195644422)

[**B.** **WHEN NOT TO USE TRIGGERS** 9](#_Toc195644423)

[**C.** **BEST PRACTICES** 9](#_Toc195644424)

[**VI.** **POSITIVE IMPACTS OF TRIGGERS ON DATABASE** 10](#_Toc195644425)

[**1.** **POSITVE IMPACTS** 10](#_Toc195644426)

[**2. MONITORING AND OPTIMIZING TRIGGER USAGE** 11](#_Toc195644427)

[**VII.** **NEGATIVE IMPACTS OF TRIGGERS ON DATABASES** 11](#_Toc195644428)

[**A.** **NEGATIVE IMPACT ON DATABASES:** 11](#_Toc195644429)

[**B.** **PRACTICAL EXAMPLES** 12](#_Toc195644430)

[**1.** **PERFORMANCE OVERHEAD:** 12](#_Toc195644431)

[**2.** **CASCADE EFFECTS:** 12](#_Toc195644432)

[**3.** **DATA INTEGRITY ISSUES** 14](#_Toc195644433)

[**VIII.** **CONCLUSION** 14](#_Toc195644434)

# **INTRODUCTION TO TRIGGERS IN POSTGRESQL**

## **BRIEF OVERVIEW OF TRIGGERS AND THEIR ORIGINS**

Triggers are a powerful feature in database management systems, particularly in PostgreSQL, that automate certain actions in response to specific events occurring on a table or view. They serve as a way to enforce business rules, maintain data integrity, and perform automated tasks without requiring explicit commands from users or applications.

### **Origins of triggers**

The concept of triggers was first introduced in the late 1970s by Dr. Eugene Wong and his colleagues during the development of the relational database model at the University of California, Berkeley. The goal was to create a mechanism that could automatically enforce business rules and data integrity constraints without requiring manual intervention from users or applications.

The formal implementation of triggers became popular in the 1980s as relational database systems, such as IBM’s DB2, adopted this functionality. The term "trigger" was coined due to the mechanism's ability to "trigger" specific actions automatically when particular conditions or events were met, providing a powerful tool for managing complex database operations.

### **What are triggers?**

A trigger is essentially a function that is automatically executed (or "fired") when a defined event occurs in the database, such as inserting, updating, or deleting data. In PostgreSQL, triggers can be set up to respond to one or several types of events, and they can run before or after these events take place.

### **Key Characteristics of Triggers**

1. **Automatic Execution**: Triggers operate without manual intervention by monitoring specified data changes, allowing for seamless automation of processes.
2. **Event-Driven**: They are tied to specific events (e.g., INSERT, UPDATE, DELETE), making them suitable for actions that need to be performed conditionally based on data modifications.
3. **User-Defined Logic**: Triggers can execute user-defined functions written in languages such as PL/pgSQL. This flexibility allows developers to create complex logic that goes beyond standard SQL operations.
4. **Row-Level and Statement-Level**: Triggers can be defined to execute either for each affected row (row-level) or once per statement (statement-level), allowing for granular control over when they activate.

### **Why Do we Use Triggers?**

Triggers are beneficial for several reasons:

* **Data Integrity**: They help enforce rules and constraints at the database level, ensuring that data adheres to business logic and remains consistent.
* **Auditing and Logging**: Triggers can automatically log changes to data, making it easier to maintain audit trails and monitor data modifications over time.
* **Complex Validations**: Triggers can encapsulate complex validation logic that can be cumbersome to implement through application code or standard SQL constraints.

In the realm of database management, triggers represent a powerful and essential feature that significantly enhances the functionality and automation of data processes. Specifically, in PostgreSQL, triggers are specialized stored procedures designed to execute automatically in response to specific events occurring on database tables or views. This essay delves into the various aspects of triggers, including their definition, purpose, events that activate them, their types, advantages, and associated challenges.

# **TYPES OF TRIGGERS**

A trigger is a set of actions that the database should automatically execute whenever a certain type of operation is performed.

* Triggers are used to ; Validate data , Enforce data integrity, Automate tasks & Log changes.
* Those types of operation include INSERT , UPDATE, DELETE, CREATE, ALTER, DROP . And a trigger can be executed before or after any of these operations depending on the type of trigger. The different types of triggers are ;

## **DML TRIGGERS**

DML (Data Manipulation Language) triggers are fired in response to INSERT, UPDATE, or DELETE operations on a table. They can be used to enforce data integrity, validate data, or perform additional actions.

The subtypes of DML triggers are;

* Row-level triggers: Fired once for each row affected by the operation.
* Statement-level triggers: Fired once for the entire statement, regardless of the number of rows affected.
* BEFORE triggers: Fired before the operation is executed.
* AFTER triggers: Fired after the operation has been executed.

## **DDL TRIGGERS**

DDL (Data Definition Language) triggers are fired in response to changes to the database schema, such as CREATE, ALTER, or DROP operations. However, PostgreSQL does not directly support DDL triggers in the same way as some other databases. Instead,so i can use event triggers.

## **EVENT TRIGGERS**

Event triggers are a type of trigger that can be used to capture DDL events, such as CREATE, ALTER, or DROP operations. They were introduced in PostgreSQL 9.3

## **INSTEAD OF TRIGGERS**

INSTEAD OF triggers are used to replace the default action of an INSERT, UPDATE, or DELETE operation on a view. They allow you to specify custom actions to be taken instead of the default operation.

# **TRIGGER FUNCTIONS**

## **REQUIREMENTS OF TRIGGER FUNCTIONS IN POSTGRESQL**

1. **Function Name**: The trigger function name must be unique within the database.
2. **Return Type**: Trigger functions must return a specific type, which depends on the type of trigger:

* RETURNS TRIGGER: For row-level triggers.
* RETURNS EVENT\_TRIGGER: For event-level triggers.

1. **Language**: Trigger functions can be written in various languages, including PL/pgSQL, SQL, and C.
2. **Trigger Definition**: A trigger function must be associated with a trigger definition, which specifies the event that triggers the function.

## **STRUCTURE OF TRIGGER FUNCTIONS IN POSTGRESQL**

1. **Function Body**: The function body contains the code that will be executed when the trigger is activated.
2. **Variables**: Trigger functions can access various variables, including:

- NEW: The new row being inserted or updated.

- OLD: The old row being updated or deleted.

- TG\_TABLE\_NAME: The name of the table that triggered the function.

- TG\_OPNAME: The name of the operation that triggered the function (e.g., INSERT, UPDATE, DELETE).

1. **Return Value**: The return value of a trigger function depends on the type of trigger:

- Row-level triggers: The function should return the NEW row if the operation is to proceed, or NULL to cancel the operation.

- Event-level triggers: The function does not return a value.

**Return Type**

The return type of a trigger function depends on the type of trigger:

1. RETURNS TRIGGER: For row-level triggers, the function should return the NEW row if the operation is to proceed, or NULL to cancel the operation.

2. RETURNS EVENT\_TRIGGER: For event-level triggers, the function does not return a value.

**Example of a Trigger Function**

CREATE OR REPLACE FUNCTION log\_insert()

RETURNS TRIGGER AS $$

BEGIN

INSERT INTO log\_table (table\_name, operation, timestamp)

VALUES (TG\_TABLE\_NAME, 'INSERT', NOW());

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

In this example, the trigger function log\_insert() returns the NEW row, allowing the insert operation to proceed.

**Special Variables**

PL/pgSQL provides several special variables that can be used in trigger functions, including:

1. **NEW**: The new row being inserted or updated.
2. **OLD**: The old row being updated or deleted.
3. **TG\_TABLE\_NAME**: The name of the table that triggered the function.
4. **TG\_OPNAME**: The name of the operation that triggered the function (e.g., INSERT, UPDATE, DELETE).

## **CODE SNIPPETS**

**Example 1: Logging changes to a table**

CREATE OR REPLACE FUNCTION log\_changes()

RETURNS TRIGGER AS $$

BEGIN

INSERT INTO log\_table (table\_name, operation, timestamp)

VALUES (TG\_TABLE\_NAME, TG\_OPNAME, NOW());

RETURN NEW;

END;

$$ LANGUAGE plpgsql;

**Example 2: Validating data before inserting or updating a table**

CREATE OR REPLACE FUNCTION validate\_data()

RETURNS TRIGGER AS $$

BEGIN

IF NEW.column\_name < 0 THEN

RAISE EXCEPTION 'Invalid value';

END IF;

RETURN NEW;

END;

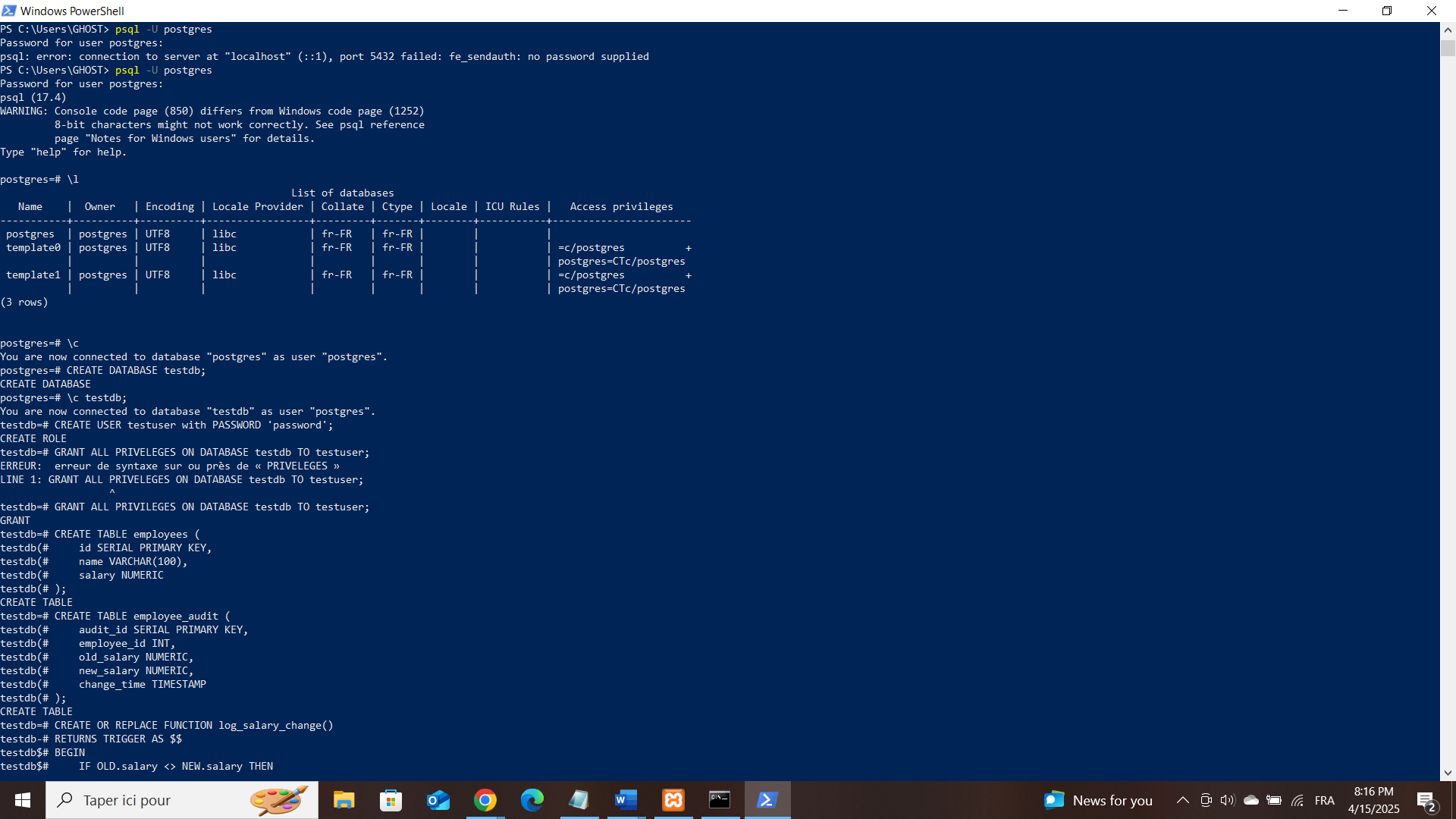
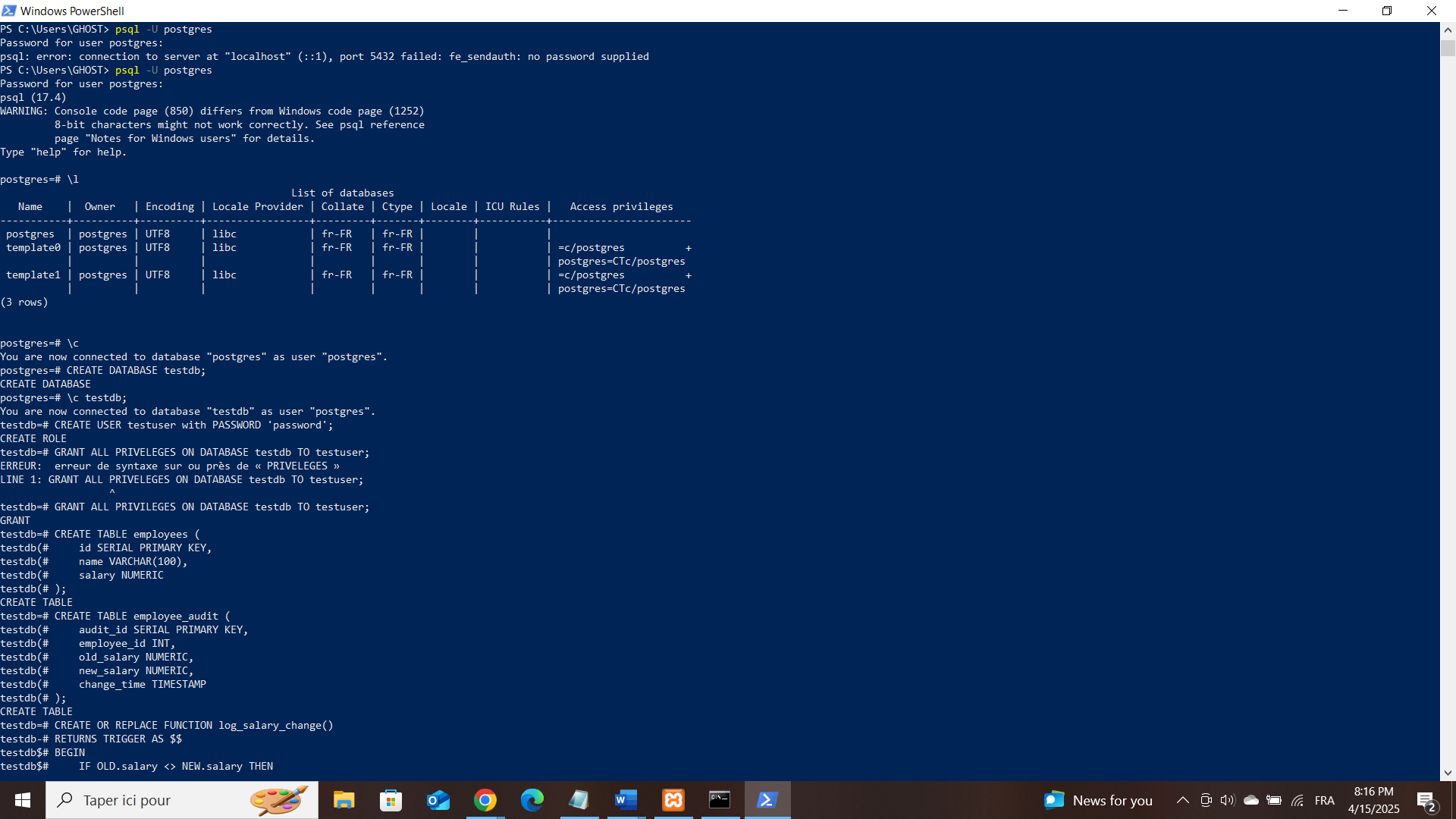
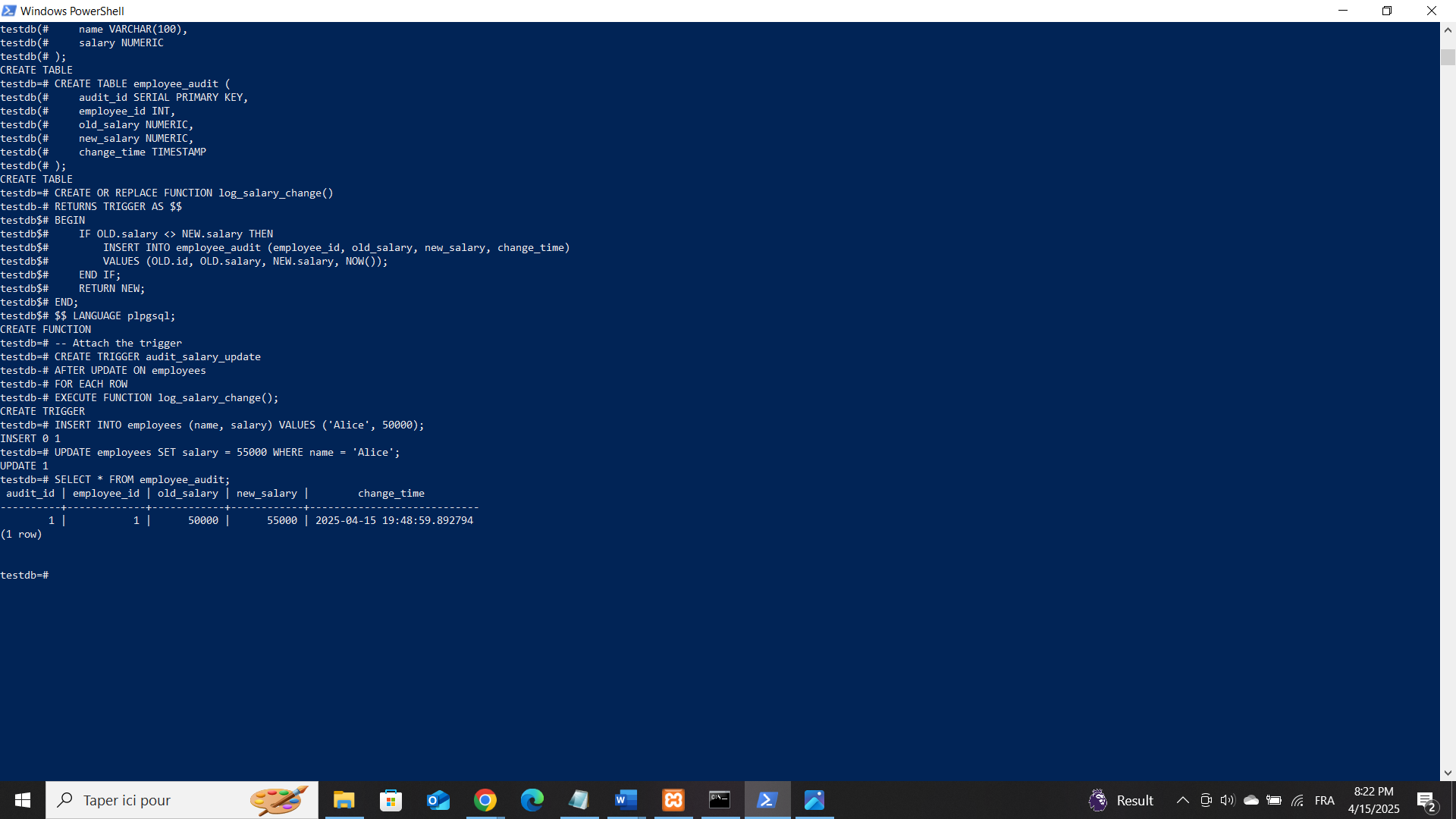
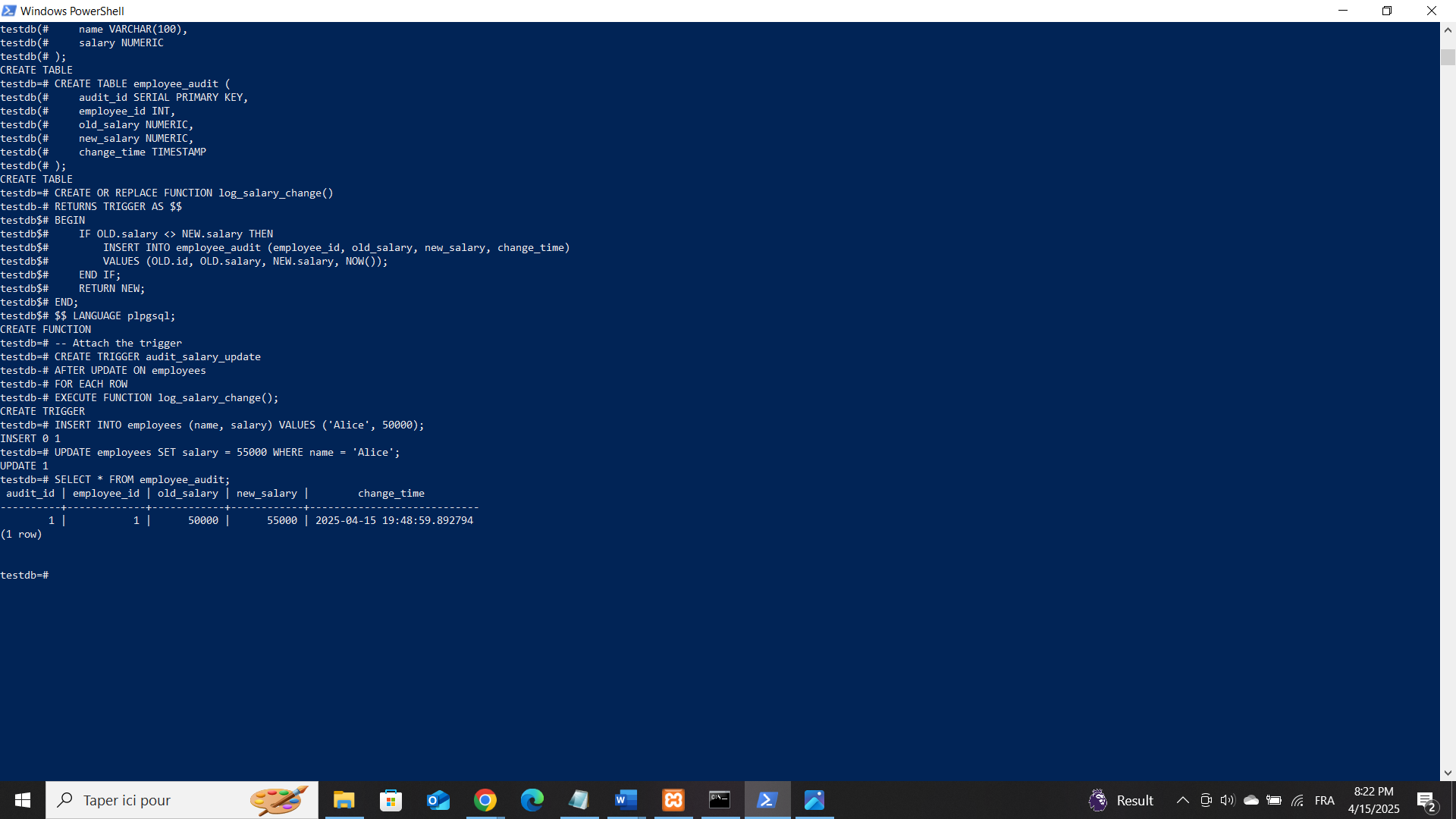
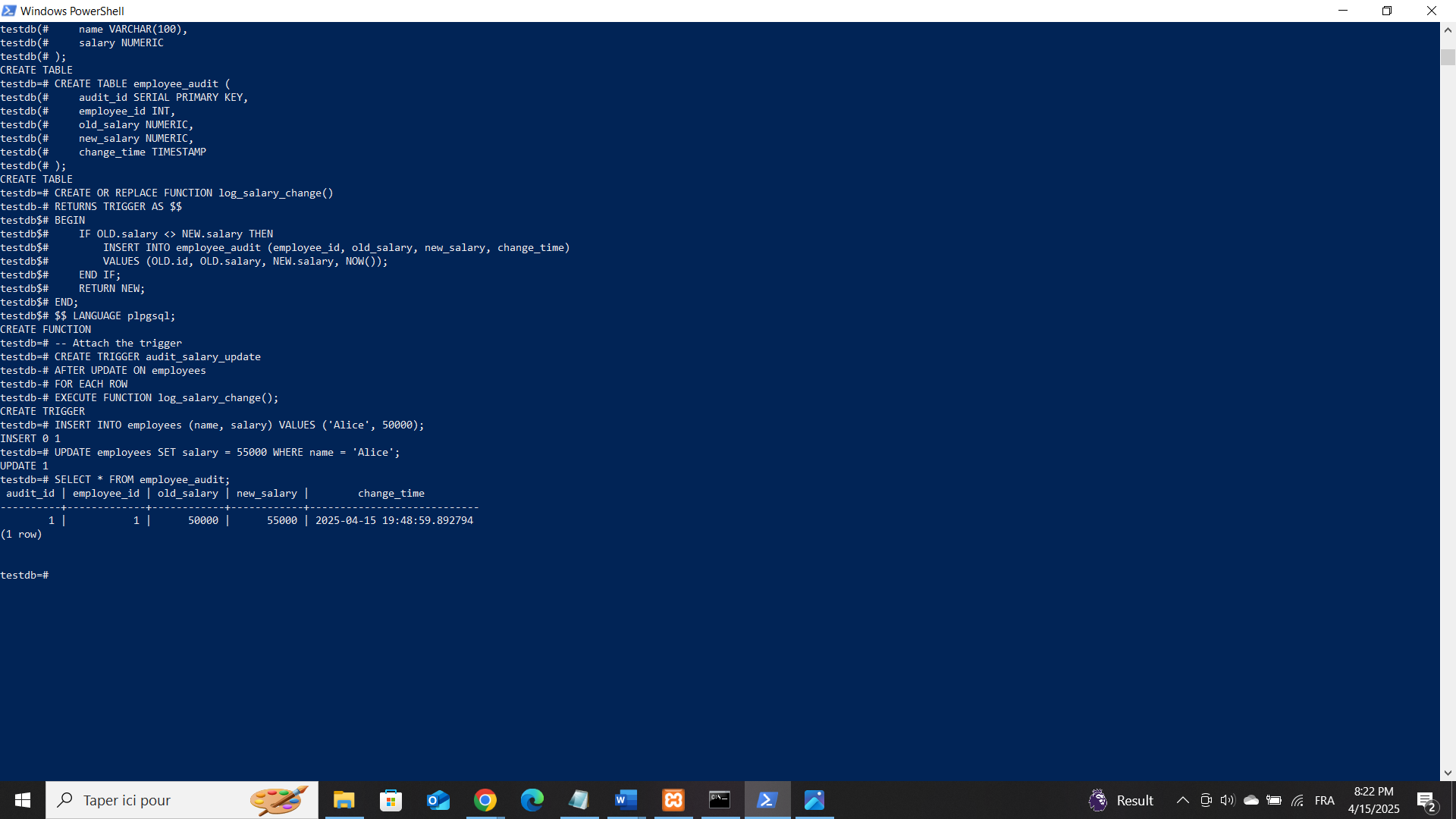
$$ LANGUAGE plpgsql;

# **USE CASES AND ILLUSTRATIONS**

## **EXAMPLE: USE CASE OF AUDIT LOGGING:**

### **What Problem Does This Solve?**

Imagine a company’s employees table stores salary data. If someone updates a salary, there’s no built-in way to track who made the change, when it happened, or what the previous value was.  
**Triggers automate this tracking**, ensuring every change is logged without manual intervention.

* STEP 1: CREATE DATABASE.
* STEP 2: CREATE employee and employee\_audit Table.
* STEP 3: CREATE TRIGGER FUNCTION
* STEP 4: ATTACH THE TRIGGER
* STEP 5: TEST THE TRIGGER

**Key Commands in psql**

| **Command** | **Description** |
| --- | --- |
| \q | Quit psql |
| \l | List all databases |
| \c dbname | Connect to a database |
| \dt | List tables |
| \d+ tablename | Show table structure |
| \e | Open the query editor |
| \? | List all psql commands |

# **BEST PRACTICES**

## **WHEN TO USE TRIGGERS**

Triggers are best used when:

* -Business rules need to be enforced automatically in the database.
* -You want to audit or log changes (e.g., track who modified data and when).
* -Complex logic that must be applied regardless of how data is modified (API, admin panel, script).
* -Maintaining historical data, such as keeping a version of each row on update.
* **Example**: Automatically inserting a row into a log table when a user is deleted

## **WHEN NOT TO USE TRIGGERS**

Prefer constraints or application logic when:

* The logic is simple and can be handled with a CHECK, UNIQUE, or FOREIGN KEY constraint.
* You want better visibility and easier debugging (triggers can make bugs harder to track).
* The operation is performance-sensitive and occurs very frequently.
* **Example**: Use a NOT NULL constraint instead of a trigger to prevent missing values.

## **BEST PRACTICES**

1. **Keep Trigger Logic Simple**

-Avoid complex calculations or heavy queries in triggers.

-Keep the function focused and readable.

1. **Use Row-level Triggers Wisely**

Row-level triggers fire once for every row, which can slow down large batch operations.

-Use statement-level triggers if you don’t need row-specific logic.

1. **Name Triggers Clearly**

Use consistent naming, like: trg\_tablename\_event\_timing

e.g., trg\_users\_insert\_after

1. **Always Document Your Triggers**

-Add comments or use metadata tables to document what each trigger does.

-Include documentation in your codebase for developers.

1. **Use the WHEN Clause to Limit Execution**

Prevent unnecessary execution using a condition:

WHEN (NEW.status IS DISTINCT FROM OLD.status)

1. **F. Test Extensively**

Simulate insert/update/delete actions and validate expected behavior.

Especially test corner cases.

1. **COMMON PITFALLS AND CHALLENGES**
2. **Performance Issues**

Triggers can degrade performance, especially on large data operations.

They can be hidden bottlenecks, as they are not visible in the SQL layer.

1. **Debugging Difficulty**

Triggers run in the background, so issues are harder to trace.

Errors in triggers can silently fail or stop transactions.

1. **Recursive Behavior**

Be cautious: one trigger can activate another (or itself), causing loops.

1. **Portability Concerns**

Triggers may be written in PL/pgSQL, which is PostgreSQL-specific.

Moving to another DBMS could require rewriting logic.

# **POSITIVE IMPACTS OF TRIGGERS ON DATABASE**

## **POSITVE IMPACTS**

* **Automatic Actions:**

Triggers run automatically when something happens (like insert, update, or delete), so you don’t need to run extra code manually.

* **Faster Data Integrity:**

They help keep data accurate and consistent without needing extra application logic, which can save time and processing.

* **Less Code Duplication:**

Business rules or checks don’t have to be repeated in many places—they can be handled once in a trigger, reducing overhead.

* **Immediate Response:**

Triggers act instantly, meaning important tasks (like logging or validation) happen right away without delay.

* **Reduced Client Load:**

Since triggers are run on the database side, the application (client) does less work, improving overall system performance.

## **2. MONITORING AND OPTIMIZING TRIGGER USAGE**

* **Use Logging Tools:**

Use database logs or monitoring tools to track how often triggers are fired and how long they take.

* **Keep Triggers Simple:**

Avoid heavy or complex logic in triggers to prevent slowing down the database.

* **Limit Number of Triggers:**

Don’t overload a table with too many triggers—they all run in order and can slow things down.

* **Avoid Recursive Triggers:**

Make sure triggers don’t unintentionally call themselves in a loop, which can cause performance issues.

**• Test and Analyze:**

Regularly test how triggers affect performance during high usage, and adjust them if needed.

# **NEGATIVE IMPACTS OF TRIGGERS ON DATABASES**

## **NEGATIVE IMPACT ON DATABASES:**

1. **Performance Overhead**: Triggers can slow down database performance, especially if they are complex or fire frequently.
2. **Increased Complexity**: Triggers can add complexity to the database, making it harder to understand and maintain.
3. **Unintended Consequences**: Triggers can have unintended consequences, such as updating or deleting data in unexpected ways.
4. **Difficulty in Testing**: Triggers can make it challenging to test database applications, as their behavior may depend on specific events or conditions.
5. **Impact on Scalability**: Triggers can impact database scalability, especially if they are not optimized for performance.

## **PRACTICAL EXAMPLES**

### **PERFORMANCE OVERHEAD:**

Triggers can slow down database performance, especially if they are complex or fire frequently.

**Example:** Suppose we have a table called "Orders" and a trigger that updates the "Total Cost" column whenever a new order item is inserted.

|  |  |
| --- | --- |
| **OrderID** | **Total Cost** |
| 1 | 100 |
| 2 | 200 |

**Trigger: Whenever a new order item is inserted, update the "Total Cost" column.**

CREATE TRIGGER UpdateTotalCost

AFTER INSERT ON OrderItems

FOR EACH ROW

BEGIN

UPDATE Orders

SET TotalCost = TotalCost + NEW.ItemPrice

WHERE OrderID = NEW.OrderID;

END;

**Diagram:**

|  |
| --- |
| Trigger |
| (UpdateTotalCost) |

|  |
| --- |
| OrderItems |

|  |
| --- |
| Orders |

Update

Insert

In this example, the trigger updates the "Total Cost" column for each new order item inserted. If there are many order items being inserted, the trigger can cause significant performance overhead.

### **CASCADE EFFECTS:**

Triggers can cause cascade effects, where one trigger fires another, leading to unintended consequences.

**Example:**

Suppose we have two tables, "Orders" and "Inventory", and two triggers:

**Trigger 1: Whenever an order is placed, update the inventory levels.**

CREATE TRIGGER UpdateInventory

AFTER INSERT ON Orders

FOR EACH ROW

BEGIN

UPDATE Inventory

SET Quantity = Quantity - NEW.Quantity

WHERE ProductID = NEW.ProductID;

END;

**Trigger 2: Whenever the inventory level falls below a certain threshold, send a notification to the supplier.**

CREATE TRIGGER NotifySupplier

AFTER UPDATE ON Inventory

FOR EACH ROW

BEGIN

IF NEW.Quantity < 10 THEN

INSERT INTO Notifications (Message)

VALUES ('Low inventory level for product ' || NEW.ProductID);

END IF;

END;

Diagram:

|  |
| --- |
| Orders |

|  |
| --- |
| Notifications |

|  |
| --- |
| Trigger2 |
| NotifySupplier |

Insert

Insert

Update

|  |
| --- |
| Trigger1 |
| UPdateInventory |

Update

|  |
| --- |
| Inventory |

In this example, the first trigger updates the inventory levels when an order is placed, which in turn fires the second trigger to send a notification to the supplier if the inventory level falls below a certain threshold. This cascade effect can lead to unintended consequences if not properly managed.

### **DATA INTEGRITY ISSUES**

Triggers can sometimes compromise data integrity if not designed or implemented correctly.

**Example**: Suppose we have a trigger that updates the "Total Cost" column whenever a new order item is inserted, but the trigger does not account for discounts or taxes.

CREATE TRIGGER UpdateTotalCost

AFTER INSERT ON OrderItems

FOR EACH ROW

BEGIN

UPDATE Orders

SET TotalCost = TotalCost + NEW.ItemPrice

WHERE OrderID = NEW.OrderID;

END;

In this example, the trigger updates the "Total Cost" column without considering discounts or taxes, which can lead to inaccurate calculations and compromise data integrity. Taking steps to mitigate them, developers can use triggers effectively to automate complex logic and improve database functionality.

### **CONCLUSION**

In summary, triggers in PostgreSQL play a critical role in enhancing database functionality by allowing automated, event-driven actions. They offer significant advantages in terms of maintaining data integrity, automating workflows, and enforcing business rules. However, careful consideration and management are necessary to avoid potential pitfalls associated with their use, such as performance overhead and increased complexity. Understanding triggers is essential for developers and database administrators looking to leverage PostgreSQL's capabilities effectively.