Triggers

Definition:

In PostgreSQL, a trigger is a special kind of stored procedure that automatically executes (or "fires") when certain events occur on a specified table or view. A trigger is bound to a specific table or view and is invoked in response to operations such as INSERT, UPDATE, or DELETE. Triggers are often used to enforce data integrity, automate tasks, or log changes to a table.

A trigger consists of two main parts:

1. **Trigger Function**: This is the function (stored procedure) that contains the logic you want to execute when the trigger fires.
2. **Trigger Event**: This is the specific action (such as INSERT, UPDATE, or DELETE) that will cause the trigger to execute.

Purpose of Triggers:

- **Data Integrity Enforcement**: Triggers can enforce business rules, validate data, and ensure that certain conditions are met before any changes are committed to the database. For example, a trigger can check that certain fields are not NULL before allowing an insert operation.

- **Automated Auditing/Logging**: Triggers can be used to automatically log changes to tables (such as logging each update or delete operation) in an audit trail table. This helps track changes for compliance or troubleshooting.

- **Cascading Actions**: Triggers can perform cascading updates or deletions. For example, when a row in a parent table is deleted, a trigger can delete related rows in child tables.

- **Preventing Invalid Operations**: Triggers can prevent operations that would violate constraints or business rules. For instance, a trigger could be used to block any insert that doesn't satisfy certain validation conditions.

- **Complex Defaults**: Triggers can be used to set default values or update values in columns automatically. For example, a trigger can automatically populate a created_at column with the current timestamp when a new row is inserted.

- **Performance Optimization**: Triggers can be used for various maintenance tasks like updating aggregate values or managing caches in real-time.

Types of Triggers:

There are several types of triggers in PostgreSQL, depending on the event that causes them to fire and whether the trigger occurs before or after the action.

1. **BEFORE Trigger**:
   - Fires before the specified event (INSERT, UPDATE, DELETE) occurs on the table.
   - Can be used to modify or validate data before it is committed to the database.
2. **AFTER Trigger**:
   - Fires after the specified event (INSERT, UPDATE, DELETE) occurs on the table.
   - Useful for operations that should happen only after the change is successfully committed (like logging, sending notifications, or updating other tables).
3. **INSTEAD OF Trigger**:
   - Used primarily with views. Fires instead of the specified event.
   - For example, when a row is INSERTed into a view, an INSTEAD OF trigger can be used to handle the operation in a different way, such as redirecting it to an underlying table.
4. **Row-Level Trigger**:
   - Executes for each row affected by the event.
   - Useful for operations like row-by-row validation or computation.
5. **Statement-Level Trigger**:
   - Executes once for the entire statement, regardless of how many rows are affected by the operation.
   - Useful for operations that should be performed once per statement, such as logging changes or updating summary tables.
6. **Trigger for Specific Event**:
   - You can define triggers for specific events like:
     - **INSERT**: Trigger when a new row is inserted.
     - **UPDATE**: Trigger when an existing row is updated.
     - **DELETE**: Trigger when a row is deleted.

Basic Syntax for Creating Triggers:

To create a trigger in PostgreSQL, you need to use the CREATE TRIGGER statement. It requires a trigger function that defines the action to be taken when the trigger is fired.

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF } { INSERT | UPDATE | DELETE }
ON table_name
[ FOR EACH ROW | FOR EACH STATEMENT ]
EXECUTE FUNCTION function_name(arguments);
```

Parameters:

- **trigger_name**: The name of the trigger.

- **BEFORE | AFTER | INSTEAD OF**: Specifies when the trigger should be executed (before, after, or instead of the event).
- **INSERT | UPDATE | DELETE**: The event that fires the trigger.
- **table_name**: The table (or view) on which the trigger operates.
- **FOR EACH ROW | FOR EACH STATEMENT**: Specifies if the trigger should fire once per row (row-level) or once per statement (statement-level).
- **function_name(arguments)**: The trigger function to execute, which contains the logic you want to perform. You can pass arguments to the function.

Trigger Timing:

- **BEFORE**: The trigger function is executed before the event (insert, update, or delete).
- **AFTER**: The trigger function is executed after the event.
- **INSTEAD OF**: The trigger function is executed instead of the event. This is typically used with views.

Trigger Events:

Triggers can be fired in response to the following events:

- **INSERT**: Fired when a new row is inserted.
- **UPDATE**: Fired when a row is updated.
- **DELETE**: Fired when a row is deleted.

Trigger Functions:

A trigger function is a function that contains the logic to be executed when the trigger is fired. The trigger function must be written in a procedural language like PL/pgSQL or SQL.

Trigger Execution Context:

OLD and NEW Records

- **OLD**: Refers to the values of the row before the event (e.g., before an update).
- **NEW**: Refers to the values of the row after the event (e.g., after an update or insert).

Example 1: Trigger on INSERT

```
-- Step 1: Create a function that will be called by the trigger
CREATE OR REPLACE FUNCTION set_created_at()
RETURNS TRIGGER AS $$
BEGIN
   NEW.created_at := NOW();  -- Set the 'created_at' column to the current timestamp
   RETURN NEW;  -- Return the new row with the updated 'created_at'
END;
$$ LANGUAGE plpgsql;

-- Step 2: Create a trigger that calls the function before INSERT
CREATE TRIGGER set_created_at_trigger
BEFORE INSERT ON employees
```

```
FOR EACH ROW
EXECUTE FUNCTION set_created_at();

-- Usage
INSERT INTO employees (name, salary) VALUES ('John Doe', 50000);
-- The 'created_at' field will be automatically set to the current timestamp
```

Explanation:

- Trigger Function: The set_created_at function assigns the current timestamp (NOW()) to the created_at column whenever a new row is inserted.
- Trigger: The trigger fires before the INSERT operation and sets the created_at column.

Example 2: Trigger on UPDATE

```
-- Step 1: Create a function that will be called by the trigger
CREATE OR REPLACE FUNCTION update_timestamp()
RETURNS TRIGGER AS $$
BEGIN
   NEW.updated_at := NOW();  -- Update the 'updated_at' column with the current timestamp
   RETURN NEW;  -- Return the updated row
END;
$$ LANGUAGE plpgsql;

-- Step 2: Create a trigger that calls the function before UPDATE
CREATE TRIGGER update_timestamp_trigger
BEFORE UPDATE ON employees
FOR EACH ROW
WHEN (OLD.salary IS DISTINCT FROM NEW.salary)  -- Only fire if salary changes
EXECUTE FUNCTION update_timestamp();

-- Usage
UPDATE employees SET salary = 55000 WHERE name = 'John Doe';
-- The 'updated_at' field will be automatically updated to the current timestamp
```

Explanation:

- Trigger Function: The update_timestamp function updates the updated_at column whenever an employee's salary is updated.
- Trigger: The trigger fires before the UPDATE operation and only triggers if the salary has changed (checked using WHEN clause).

Example 3: Trigger on DELETE

```
-- Step 1: Create an audit table to log deleted employees
CREATE TABLE employee_audit (
   audit_id SERIAL PRIMARY KEY,
   employee_id INT,
    name TEXT,
    salary DECIMAL,
    deleted_at TIMESTAMP
);

-- Step 2: Create a function to log the deleted employee's data
CREATE OR REPLACE FUNCTION log_employee_deletion()
RETURNS TRIGGER AS $$
BEGIN
    -- Log the deleted employee's data into the audit table
    INSERT INTO employee_audit (employee_id, name, salary, deleted_at)
    VALUES (OLD.id, OLD.name, OLD.salary, NOW());
    RETURN OLD;  -- Return the old row (since it's a DELETE trigger)
END;
$$ LANGUAGE plpgsql;

-- Step 3: Create a trigger that calls the function after DELETE
CREATE TRIGGER log_employee_deletion_trigger
AFTER DELETE ON employees
FOR EACH ROW
EXECUTE FUNCTION log_employee_deletion();

-- Usage
DELETE FROM employees WHERE name = 'John Doe';
-- The deleted employee's data will be logged in the 'employee_audit' table
```

Explanation:

- Audit Table: The employee_audit table is used to store logs of deleted employee records.
- Trigger Function: The log_employee_deletion function inserts the deleted employee's data into the employee_audit table.
- Trigger: The trigger fires after the DELETE operation and logs the deleted employee’s data.

Example 4: Trigger on INSERT, UPDATE, and DELETE

```
-- Step 1: Create a general audit table
CREATE TABLE employee_audit (
    audit_id SERIAL PRIMARY KEY,
    employee_id INT,
    name TEXT,
    salary DECIMAL,
    operation TEXT,
    action_time TIMESTAMP
);
```

```
-- Step 2: Create a function to log all changes
CREATE OR REPLACE FUNCTION audit_employee_changes()
RETURNS TRIGGER AS $$
BEGIN
   IF TG_OP = 'INSERT' THEN
      INSERT INTO employee_audit (employee_id, name, salary, operation, action_time)
      VALUES (NEW.id, NEW.name, NEW.salary, 'INSERT', NOW());
   ELSIF TG_OP = 'UPDATE' THEN
      INSERT INTO employee_audit (employee_id, name, salary, operation, action_time)
      VALUES (NEW.id, NEW.name, NEW.salary, 'UPDATE', NOW());
   ELSIF TG_OP = 'DELETE' THEN
      INSERT INTO employee_audit (employee_id, name, salary, operation, action_time)
      VALUES (OLD.id, OLD.name, OLD.salary, 'DELETE', NOW());
   END IF;
   RETURN NULL;  -- Return NULL because it's a row-level trigger
END;
$$ LANGUAGE plpgsql;

-- Step 3: Create a trigger that fires on INSERT, UPDATE, or DELETE
CREATE TRIGGER audit_employee_changes_trigger
AFTER INSERT OR UPDATE OR DELETE ON employees
FOR EACH ROW
EXECUTE FUNCTION audit_employee_changes();

-- Usage
INSERT INTO employees (name, salary) VALUES ('John Doe', 50000);
UPDATE employees SET salary = 55000 WHERE name = 'John Doe';
DELETE FROM employees WHERE name = 'John Doe';
-- All actions will be logged in the 'employee_audit' table
```

Explanation:

- Trigger Function: The audit_employee_changes function logs changes (inserts, updates, and deletes) into the employee_audit table.
- Trigger: The trigger fires after any INSERT, UPDATE, or DELETE on the employees table.
- TG_OP: The TG_OP special variable determines which operation (INSERT, UPDATE, or DELETE) triggered the function.

Example 5: Trigger to Prevent Negative Salary

```
-- Step 1: Create a function that raises an exception if the salary is negative
CREATE OR REPLACE FUNCTION prevent_negative_salary()
RETURNS TRIGGER AS $$
BEGIN
```

```
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION 'Salary cannot be negative';
    END IF;
    RETURN NEW;  -- Return the new row if the salary is valid
END;
$$ LANGUAGE plpgsql;

-- Step 2: Create a trigger that calls the function before INSERT or UPDATE
CREATE TRIGGER prevent_negative_salary_trigger
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION prevent_negative_salary();

-- Usage
INSERT INTO employees (name, salary) VALUES ('Jane Doe', -5000);  -- Will raise an exception
UPDATE employees SET salary = -10000 WHERE name = 'John Doe';    -- Will raise an exception
```

Explanation:

- Trigger Function: The prevent_negative_salary function checks if the salary value is negative. If so, it raises an exception and prevents the insertion or update.
- Trigger: The trigger fires before INSERT or UPDATE on the employees table. If the salary is negative, the trigger prevents the change by raising an exception.