1. What is a Trigger in PostgreSQL?

A trigger is a database object that is linked to a table and automatically fires (executes) a trigger function when a specified event happens on that table—INSERT, UPDATE, DELETE, or TRUNCATE.

- The trigger can be set to fire BEFORE or AFTER the event.
- It can run FOR EACH ROW (row-level trigger) or FOR EACH STATEMENT (statement-level trigger).
- Triggers help maintain data integrity, audit changes, enforce business rules, or perform other automated tasks.

2. Trigger Function

A trigger function is a special type of function written in PL/pgSQL (or another supported language) that is specifically designed for triggers. It returns the special type `TRIGGER` and can access:

- `OLD` record: the row data before modification (for UPDATE and DELETE)
- `NEW` record: the row data after modification (for INSERT and UPDATE)
- Special variables such as `TG_OP` (operation type), `TG_NAME` (trigger name), and others.

3. How to Create a Trigger Function

Example syntax to define a trigger function:

```
CREATE OR REPLACE FUNCTION your_trigger_function()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    -- Trigger logic goes here
    -- For example, inserting records for audit logs
    RETURN NEW;  -- For INSERT/UPDATE triggers, return the new row
END;
$$;
```

4. How to Create a Trigger

Once the trigger function is defined, you create a trigger on a specific table that calls this function when the triggering event occurs:

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

5. Example: Auditing Updates on an Employee Table

Step 1: Create the table and audit log table

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    salary NUMERIC
);

CREATE TABLE salary_log (
    id SERIAL PRIMARY KEY,
    employee_id INT,
    old_salary NUMERIC,
    new_salary NUMERIC,
    changed_at TIMESTAMP DEFAULT current_timestamp
);
```

Step 2: Create the trigger function to log salary changes

```
CREATE OR REPLACE FUNCTION update_salary_log()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
```

```
BEGIN
    IF NEW.salary <> OLD.salary THEN
        INSERT INTO salary_log(employee_id, old_salary, 
new_salary)
        VALUES (NEW.id, OLD.salary, NEW.salary);
    END IF;
    RETURN NEW;
END;
$$;
```

Step 3: Create the trigger on the employees table

```
CREATE TRIGGER salary_update_trigger
AFTER UPDATE OF salary
ON employees
FOR EACH ROW
EXECUTE FUNCTION update_salary_log();
```

Whenever an employee's salary changes, an entry is logged automatically.

6. Important Notes on Triggers

- Use BEFORE triggers if you want to modify or validate data before it is written.
- Use AFTER triggers to perform actions after the data has been modified.
- Row-level triggers execute once per affected row.
- Statement-level triggers execute once per SQL statement, regardless of how many rows are affected.
- You can pass arguments to triggers if needed.
- Use transactions carefully when performing complex operations in trigger functions.

7. Key Trigger Variables Available Inside Trigger Functions

Variable	**Description**
`OLD`	Old row data (before UPDATE or DELETE)
`NEW`	New row data (for INSERT or UPDATE)
`TG_OP`	Trigger event type (`INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`)
`TG_NAME`	Name of the trigger
`TG_WHEN`	When the trigger fires (`BEFORE` or `AFTER`)
`TG_TABLE_NAME`	Table that the trigger is attached to
`TG_LEVEL`	`ROW` or `STATEMENT`

8. Summary Example: Basic Trigger & Function

```
-- Create a simple logging table
CREATE TABLE employee_audits (
    employee_id INT,
    old_name TEXT,
    new_name TEXT,
```

```sql
    changed_at TIMESTAMP DEFAULT current_timestamp
);

-- Trigger function to log name changes
CREATE OR REPLACE FUNCTION log_name_changes()
RETURNS TRIGGER AS $$
BEGIN
    IF OLD.last_name IS DISTINCT FROM NEW.last_name THEN
        INSERT INTO employee_audits(employee_id, old_name,
new_name)
        VALUES (NEW.id, OLD.last_name, NEW.last_name);
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Create trigger on employees table
CREATE TRIGGER last_name_change_trigger
BEFORE UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION log_name_changes();
```

1. What is a Stored Procedure in PostgreSQL?

- A stored procedure is a set of SQL statements saved in the database that can be executed by calling the procedure.
- Supports transaction control (begin, commit, rollback) inside the procedure.
- Can accept input parameters (IN, INOUT modes), but does not return a value like a function.
- Helps centralize logic on the server side for reuse, consistency, and efficiency.

2. Syntax of Creating a Stored Procedure

```sql
CREATE [OR REPLACE] PROCEDURE procedure_name(
    [ IN parameter_name parameter_type [, ...] ]
    [ INOUT parameter_name parameter_type [, ...] ]
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Procedure body: SQL statements, control structures,
transaction commands
END;
$$;
```

- `IN` parameters provide input values.
- `INOUT` parameters can be used for input/output values.
- No `OUT` parameters allowed.
- The procedure body is written in `plpgsql` or other supported procedural languages.
- Transaction statements like `COMMIT` or `ROLLBACK` can be used inside.

3. Calling a Stored Procedure

Use the `CALL` statement to execute a procedure:

```sql
CALL procedure_name(param1, param2, ...);
```

4. Simple Examples

Example 1: Create a basic procedure to insert data

Assuming a table:

```
sql
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    age INT
);
```

Create procedure to insert an employee:

```
sql
CREATE OR REPLACE PROCEDURE add_employee(emp_name VARCHAR,
emp_age INT)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO employees (name, age) VALUES (emp_name,
emp_age);
END;
$$;
```

Call it:

```
sql
CALL add_employee('John Doe', 25);
```

Example 2: Procedure with transaction control

```
sql
CREATE OR REPLACE PROCEDURE update_employee_age(emp_id INT,
new_age INT)
LANGUAGE plpgsql
AS $$
BEGIN
```

```
    UPDATE employees SET age = new_age WHERE id = emp_id;
    COMMIT;  -- explicit commit inside procedure
END;
$$;
```

5. Differences Between Functions and Procedures in PostgreSQL

Feature	**Function**	**Stored Procedure**
Can execute transaction control	No	Yes
Return value	Must return a value	No return value
Called via	`SELECT` or directly	`CALL` statement
Use cases	Calculations, scalar values	Multi-statement tasks, transactions

Functions in PostgreSQL are user-defined routines that encapsulate reusable logic, computations, or operations which can be executed on demand. They can take input parameters, perform operations (such as calculations or data manipulations), and return results.
Here's a detailed overview of PostgreSQL functions, including how to create and use them:

What Are Functions in PostgreSQL?

- User-Defined Functions (UDFs) allow you to package SQL or procedural code (e.g., PL/pgSQL) into callable operations within the database.
- Functions can accept parameters and return a value (scalar, row, or set of rows).
- They are useful for simplifying complex logic, improving code reuse, ensuring consistency, and performance optimization.
- PostgreSQL supports multiple languages for functions, such as `SQL`, `PL/pgSQL` (most common), `PL/Python`, etc.

Basic Syntax for Creating a Function

```
sql
CREATE [OR REPLACE] FUNCTION function_name(parameter1 datatype,
parameter2 datatype, ...)
RETURNS return_datatype
LANGUAGE plpgsql        -- Procedural language
AS $$
BEGIN
    -- Function logic here
    RETURN some_value;
END;
$$;
```

- Use `OR REPLACE` to modify an existing function without dropping it.
- The function body is typically written in `plpgsql`, PostgreSQL's procedural language.
- The `RETURNS` clause defines the return type.
- `parameter_name datatype` defines input parameters.

Simple Example: A Function That Adds Two Integers

```
sql
CREATE OR REPLACE FUNCTION add_numbers(x int, y int)
RETURNS int
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN x + y;
END;
$$;
```

You can call this function as:

```
sql
SELECT add_numbers(5, 3);  -- Returns 8
```

Important Concepts

- Return Types: Functions can return scalar types (`int`, `text`), composite types (rows), sets of rows (`RETURNS TABLE(...)` or `RETURNS SETOF`), or `void`.
- Parameters: Default mode is `IN`, but `INOUT` parameters are also supported.
- Variable Declaration: Inside the function, you can declare local variables in a `DECLARE` section.
- Control Structures: Use conditional statements (`IF`, `CASE`), loops (`FOR`, `WHILE`).
- Exception Handling: Use `BEGIN ... EXCEPTION ... END` blocks to catch errors.

Example: Function with Variables and Conditional Logic

```
sql
CREATE OR REPLACE FUNCTION greet_user(name TEXT)
RETURNS TEXT
LANGUAGE plpgsql
AS $$
DECLARE
    greeting TEXT;
BEGIN
    IF name IS NULL OR name = '' THEN
```

```
        RETURN 'Hello, Guest!';
    ELSE
        greeting := 'Hello, ' || name || '!';
        RETURN greeting;
    END IF;
END;
$$;
```

Call it as:

```sql
SELECT greet_user('Alice');  -- Returns "Hello, Alice!"
SELECT greet_user(NULL);     -- Returns "Hello, Guest!"
```

Calling Functions

- Use `SELECT function_name(args);` for those returning values.
- For functions returning sets, you can use them in the `FROM` clause, e.g., `SELECT * FROM your_function();`.