Lecture 12. Trigger

What is a Trigger?

- A trigger is a **database objects** that **automatically executes a function** when a specific event happens on a table
- Events: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`
- The code that actually runs is a **trigger function** (usually written in PL/pgSQL).

Idea:

> "When something happens to this table, run this code."

Why Use Triggers?

- When DB is used by **many different applications**, you want some logic to live **inside the database**, not in every app.
- Examples:
    - **Audit/history**: keep history of changes (e.g. old last names, changes log) on every `INSERT` / `UPDATE`.
    - **Automatic actions**: maintain summary tables, counters, etc.
- Key point: triggers run **automatically** whenever the configured event occurs.

Trigger Types (Row vs Statement)

PostgreSQL supports **two main trigger types**:

1. **Row-level trigger**
    - Defined with `FOR EACH ROW`.
    - Fired **once per affected row**.
    - Used when you need to inspect or modify individual rows.
2. **Statement-level trigger**
    - Defined with `FOR EACH STATEMENT`.
    - Fired **once per statement**, no matter how many rows are affected.
    - Good for logging that “something happened” or doing bulk actions.

PostgreSQL-Specific Trigger Features vs SQL Standard

PostgreSQL has some specifics:

- Fires triggers also on **`TRUNCATE`** (not just INSERT/UPDATE/DELETE).
- Allows **statement-level triggers on views** (and `INSTEAD OF` triggers on views in general).
- Requires a **user-defined function** as the trigger action; SQL standard allows writing arbitrary SQL directly in the trigger body.

Trigger Function (Must Return `TRIGGER`)

Before you create a trigger, you must define a **trigger function**:

```
CREATE FUNCTION trigger_function()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
   -- trigger logic
   RETURN NEW;  -- or RETURN OLD, depending on BEFORE/AFTER and ev
```

```
ent
END;
$$;
```

Key points:

- Declared **with no arguments** and `RETURNS TRIGGER`.
- Written in `plpgsql` (most common), but can be other languages.
- When executed as a trigger, it has special variables: PostgreSQL
  - `NEW` – new row (for `INSERT` and `UPDATE` in row-level triggers).
  - `OLD` – old row (for `UPDATE` and `DELETE` in row-level triggers).
  - `TG_TABLE_NAME`, `TG_WHEN`, `TG_LEVEL`, etc. – meta info about trigger.

CREATE TRIGGER – Syntax & Meaning

Basic form:

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

Pieces:

- `trigger_name` – name of the trigger.
- `BEFORE` / `AFTER` – **when** it fires relative to the event:
  - `BEFORE` – before the row is changed (you can modify `NEW` or cancel).
  - `AFTER` – after the change is done (good for logging).
- `event` – `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE` (or multiple with `OR` in full syntax).
- `FOR EACH ROW` – row-level; `FOR EACH STATEMENT` – statement-level.
- `EXECUTE PROCEDURE trigger_function()` – bind to the trigger function.

Example Idea from Slides (Employees Audit)

Slides show tables: `employees` and `employee_audits`

```
CREATE TABLE employees (
    id          INT GENERATED ALWAYS AS IDENTITY,
    first_name  VARCHAR(40) NOT NULL,
    last_name   VARCHAR(40) NOT NULL,
    PRIMARY KEY (id)
);

CREATE TABLE employee_audits (
    id          INT GENERATED ALWAYS AS IDENTITY,
    employee_id INT         NOT NULL,
    last_name   VARCHAR(40) NOT NULL,
    changed_on  TIMESTAMP(6) NOT NULL
);
```

Typical trigger scenario (conceptually):

- Event: `UPDATE` of `employees.last_name`.
- Action: insert a row into `employee_audits` with:
    - `employee_id` = `OLD.id`
    - `last_name` = `OLD.last_name`
    - `changed_on` = `CURRENT_TIMESTAMP`

This is a **row-level AFTER UPDATE trigger** that logs each change.

When to Use Triggers

Use triggers when you need:

- **Cross-application logic** in one place (DB), not duplicated.

- **Automatic auditing**: history of data changes.
- **Enforcing custom business rules** that are too complex for simple constraints.
- Automatic side-effects when data changes.

DROP TRIGGER

```
DROP TRIGGER [IF EXISTS] trigger_name
ON table_name
[ CASCADE | RESTRICT ];
```

- `CASCADE` – also drop dependent objects.
- `RESTRICT` – fail if dependencies exist (default).

ALTER TRIGGER

```
ALTER TRIGGER trigger_name
ON table_name
RENAME TO new_trigger_name;
```