

PL/pgSQL Triggers



12

Copyright

© Postgres Professional, 2017–2021

Authors: Egor Rogov, Pavel Luzanov

Translated by Liudmila Mantrova

Usage of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed on an unrestricted basis. Commercial use is only possible with prior written permission of Postgres Professional company. Modification of course materials is forbidden.

Contact Us

Please send your feedback to: edu@postgrespro.ru

Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Agenda

Triggers and trigger functions

Triggers' firing time

Execution context of a trigger function

Return values

Dos and don'ts when using triggers

Event triggers

A trigger

a database object: the list of events to process
once an event occurs, the trigger function is called,
and the call context is passed to this function

Trigger functions

a database object: event-processing code
is executed in the same transaction as the main operation
convention: the function does not take any parameters,
returns a value of the `trigger` pseudotype (which is virtually a record)
can be used in several triggers

Triggers enables us to perform particular actions in response to particular events. A trigger consists of two parts: the trigger itself (which defines the events) and a trigger function (which defines the actions). Both the trigger and the function are independent database objects.

When an event occurs to which the trigger is subscribed, the trigger function is called. It receives the context of the call that defines the exact trigger that has called the function and the exact conditions that have led to this call.

A trigger function is a regular function that follows some conventions:

- It can be written in any language except pure SQL.
- It must have no parameters.
- Its return value is of the `trigger` type (it is actually a pseudotype; a record corresponding to a table row is returned instead).

The trigger function is executed in the same transaction as the main operation. Thus, if a trigger function results in an error, the whole transaction is aborted.

<https://postgrespro.com/docs/postgresql/12/trigger-definition>

INSERT, UPDATE, DELETE

tables	before/after	statement
	before/after	row
views	before/after	statement
	instead of	row

TRUNCATE

tables	before/after	statement
--------	--------------	-----------

WHEN condition

sets an additional filter

Triggers can fire for INSERT, UPDATE, or DELETE operations performed on tables or views, as well as for the TRUNCATE operation on tables.

A trigger can fire before the specified action (BEFORE), after the action (AFTER), or instead of the action (INSTEAD OF).

A trigger can fire once for the whole operation (FOR EACH STATEMENT) or for each affected row (FOR EACH ROW).

There are some combinations of these conditions that are not supported. For example, INSTEAD OF triggers can be defined only for views at the row level, while TRUNCATE triggers can be defined only for tables and only at the statement level. Possible combinations are listed on this slide.

Besides, you can limit the area controlled by the trigger by specifying the WHEN condition: if this condition is false, the trigger does not fire.

<https://postgrespro.com/docs/postgresql/12/sql-createtrigger>

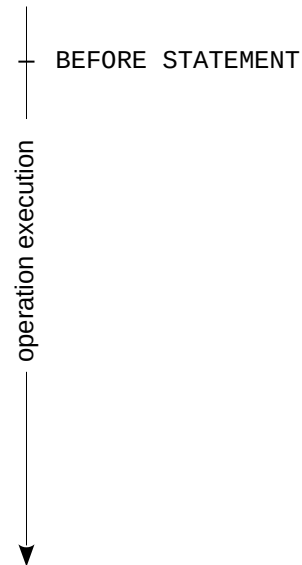
Before Statement



Is triggered
before the operation

Return value
is ignored

Context
TG variables



5

Let's take a closer look at different trigger types.

The `BEFORE STATEMENT` trigger fires only once per operation, regardless of the number of affected rows (even if there is none). It happens before the start of the operation.

The return value of the trigger function is ignored, it can simply return `NULL`. If there is an error in the trigger, the operation is canceled.

Since the trigger function has no parameters, the call context in PL/pgSQL is passed via predefined TG variables, such as:

- `TG_WHEN = "BEFORE"`
- `TG_LEVEL = "STATEMENT"`
- `TG_OP = "INSERT"/"UPDATE"/"DELETE"/"TRUNCATE"`

You can also pass a user-defined context (which is analogous to the absent parameters) via the `TG_ARGV` variable, although it is usually advisable to create several specialized functions instead of a single generic one.

<https://postgrespro.com/docs/postgresql/12/plpgsql-trigger>

Before Row

Is triggered

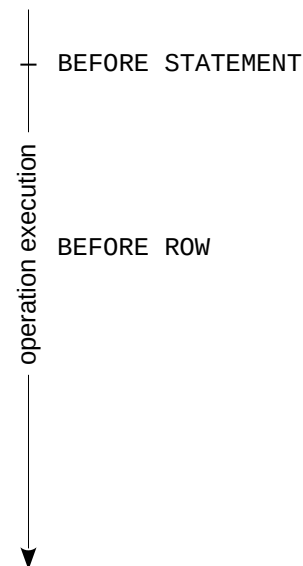
before the action on the row
during the statement execution

Return value

a row (possibly modified)
null cancels the operation

Context

OLD update, delete
NEW insert, update
TG variables



BEFORE ROW triggers fire each time an operation is about to process a row. It happens right during the operation execution.

Trigger functions get the context via variables, such as:

- OLD — an old state of the row (undefined for insertion)
- NEW — an updated state of the row (undefined for deletion)
- TG_WHEN = "BEFORE"
- TG_LEVEL = "ROW"
- TG_OP = "INSERT"/"UPDATE"/"DELETE"

The NULL return value is interpreted as cancellation of the action for the current row. The execution of the operation itself will continue, but the current row won't be processed, and other triggers won't fire for this row.

To avoid interfering with the operation, the trigger must return the received row without any modifications, so it must always return NEW for insert and update operations. For delete operations, it can return any value except NULL (usually OLD is used).

But the trigger function can also change the NEW value to affect the result of the operation; this trigger is often defined exactly for this purpose.

Instead of Row

Is triggered

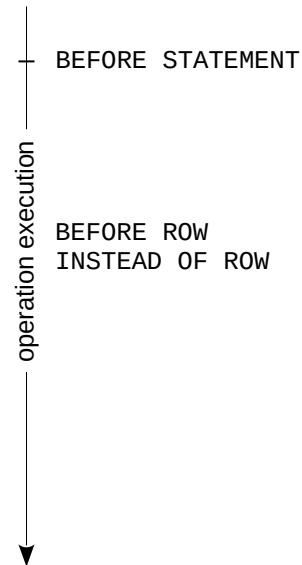
instead of the action on the row
for views

Return value

a row (possibly modified) —
is shown in RETURNING
null cancels the operation

Context

OLD update, delete
NEW insert, update
TG variables



INSTEAD OF triggers are very similar to BEFORE triggers, but they can be defined only for views and fire instead of the specified operation, not before it.

Such triggers usually perform operations on the base tables for views. The trigger can also return a modified NEW value: this value will be available when performing an operation with the RETURNING clause.

After Row

Is triggered

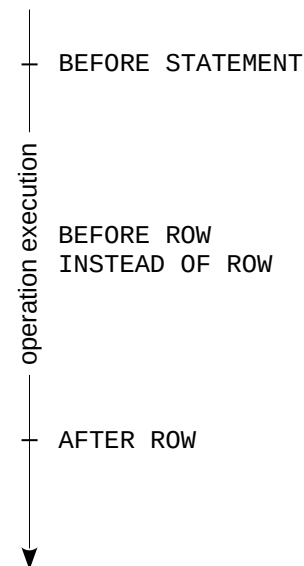
after the operation
the queue of rows satisfying the WHEN condition

Return value

is ignored

Context

OLD, OLD TABLE update, delete
NEW, NEW TABLE insert, update
TG variables



Just as BEFORE ROW, AFTER ROW triggers fire for each affected row; but it happens only after the whole operation is complete, not immediately after processing the row. For this purpose, all events are placed in a queue and processed after the operation has finished. The fewer events get queued, the smaller overhead will be incurred; that's why it is recommended to use the WHEN clause in this case, which allows us to filter out the rows that we definitely won't need.

The return value of the AFTER ROW triggers is ignored (because the operation is already complete).

The context of the trigger function is constituted by the following variables:

- OLD — an old value of the row (undefined for insertion)
- NEW — an updated value of the row (undefined for deletion)

Apart from these variables, in PostgreSQL 10 or higher, the trigger function can get access to special *transition tables*. The table listed in trigger definition as OLD TABLE contains old values of the rows processed by the trigger; NEW TABLE provides updated values of these rows.

Regular TG variables are also available, including the following ones:

- TG_WHEN = "AFTER"
- TG_LEVEL = "ROW"
- TG_OP = "INSERT"/"UPDATE"/"DELETE"

After Statement

Is triggered

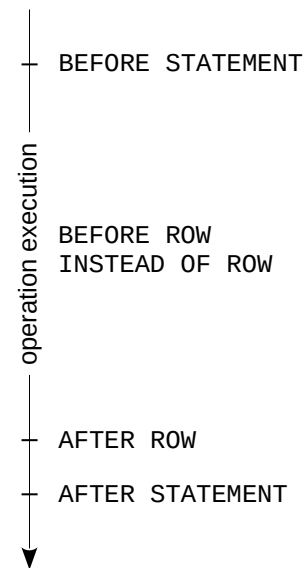
after the operation
(even if none of the rows are affected)

Return value

is ignored

Context

OLD TABLE update, delete
NEW TABLE insert, update
TG variables



The AFTER STATEMENT trigger fires after the operation has completed (including all the AFTER ROW triggers, if any). This trigger fires only once regardless of the number of the affected rows.

The return value of the trigger function is ignored.

The call context is passed using transition tables. The trigger function can access these table to analyze all the affected rows. Transition tables are usually used with AFTER STATEMENT, not with AFTER ROW triggers.

Besides, regular TG variables are defined, such as:

- TG_WHEN = "AFTER"
- TG_LEVEL = "STATEMENT"
- TG_OP = "INSERT"/"UPDATE"/"DELETE"/"TRUNCATE"

The Triggers' Firing Order

Let's create a "universal" trigger function that describes the context in which it is called. The context is passed in various TG variables.

We are going to define triggers for various events and observe the order in which the triggers are fired during execution.

```
=> CREATE OR REPLACE FUNCTION describe() RETURNS trigger
AS $$
DECLARE
    rec record;
    str text := '';
BEGIN
    IF TG_LEVEL = 'ROW' THEN
        CASE TG_OP
            WHEN 'DELETE' THEN rec := OLD; str := OLD::text;
            WHEN 'UPDATE' THEN rec := NEW; str := OLD || ' -> ' || NEW;
            WHEN 'INSERT' THEN rec := NEW; str := NEW::text;
        END CASE;
    END IF;
    RAISE NOTICE '% % % %: %',
        TG_TABLE_NAME, TG_WHEN, TG_OP, TG_LEVEL, str;
    RETURN rec;
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

A table:

```
=> CREATE TABLE t(
    id integer PRIMARY KEY,
    s text
);
```

CREATE TABLE

Triggers at the statement level:

```
=> CREATE TRIGGER t_before_stmt
BEFORE INSERT OR UPDATE OR DELETE -- events
ON t                               -- table
FOR EACH STATEMENT                -- level
EXECUTE FUNCTION describe();      -- trigger function
```

CREATE TRIGGER

```
=> CREATE TRIGGER t_after_stmt
AFTER INSERT OR UPDATE OR DELETE ON t
FOR EACH STATEMENT EXECUTE FUNCTION describe();
```

CREATE TRIGGER

Triggers at the row level:

```
=> CREATE TRIGGER t_before_row
BEFORE INSERT OR UPDATE OR DELETE ON t
FOR EACH ROW EXECUTE FUNCTION describe();
```

CREATE TRIGGER

```
=> CREATE TRIGGER t_after_row
AFTER INSERT OR UPDATE OR DELETE ON t
FOR EACH ROW EXECUTE FUNCTION describe();
```

CREATE TRIGGER

Let's perform an insert operation:

```
=> INSERT INTO t VALUES (1, 'aaa');
```

```
NOTICE: t BEFORE INSERT STATEMENT:
NOTICE: t BEFORE INSERT ROW: (1,aaa)
NOTICE: t AFTER INSERT ROW: (1,aaa)
NOTICE: t AFTER INSERT STATEMENT:
INSERT 0 1
```

And now run an update operation:

```
=> UPDATE t SET s = 'bbb';

NOTICE:  t BEFORE UPDATE STATEMENT:
NOTICE:  t BEFORE UPDATE ROW: (1,aaa) -> (1,bbb)
NOTICE:  t AFTER UPDATE ROW: (1,aaa) -> (1,bbb)
NOTICE:  t AFTER UPDATE STATEMENT:
UPDATE 1
```

Statement-level triggers will fire even if the command has not processed any rows at all:

```
=> UPDATE t SET s = 'ccc' where id = 0;

NOTICE:  t BEFORE UPDATE STATEMENT:
NOTICE:  t AFTER UPDATE STATEMENT:
UPDATE 0
```

Here is a subtle point: the INSERT statement with the ON CONFLICT clause activates BEFORE triggers both on inserts and updates:

```
=> INSERT INTO t VALUES (1,'ccc'), (3,'ddd')
ON CONFLICT(id) DO UPDATE SET s = EXCLUDED.s;

NOTICE:  t BEFORE INSERT STATEMENT:
NOTICE:  t BEFORE UPDATE STATEMENT:
NOTICE:  t BEFORE INSERT ROW: (1,ccc)
NOTICE:  t BEFORE UPDATE ROW: (1,bbb) -> (1,ccc)
NOTICE:  t BEFORE INSERT ROW: (3,ddd)
NOTICE:  t AFTER UPDATE ROW: (1,bbb) -> (1,ccc)
NOTICE:  t AFTER INSERT ROW: (3,ddd)
NOTICE:  t AFTER UPDATE STATEMENT:
NOTICE:  t AFTER INSERT STATEMENT:
INSERT 0 2
```

And finally, let's try out deletion:

```
=> DELETE FROM t;

NOTICE:  t BEFORE DELETE STATEMENT:
NOTICE:  t BEFORE DELETE ROW: (1,ccc)
NOTICE:  t BEFORE DELETE ROW: (3,ddd)
NOTICE:  t AFTER DELETE ROW: (1,ccc)
NOTICE:  t AFTER DELETE ROW: (3,ddd)
NOTICE:  t AFTER DELETE STATEMENT:
DELETE 2
```

Transition Tables

Let's create a trigger function that shows the contents of transition tables. Here we use `old_table` and `new_table` names: they will be declared as part of the trigger definition.

Transition tables look just like regular ones, but they are not included into the system catalog and are located in RAM (although they can be flushed to disk if they get too large).

```
=> CREATE OR REPLACE FUNCTION transition() RETURNS trigger
AS $$
DECLARE
    rec record;
BEGIN
    IF TG_OP = 'DELETE' OR TG_OP = 'UPDATE' THEN
        RAISE NOTICE 'Old state: ';
        FOR rec IN SELECT * FROM old_table LOOP
            RAISE NOTICE '%', rec;
        END LOOP;
    END IF;
    IF TG_OP = 'UPDATE' OR TG_OP = 'INSERT' THEN
        RAISE NOTICE 'New state: ';
        FOR rec IN SELECT * FROM new_table LOOP
            RAISE NOTICE '%', rec;
        END LOOP;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION
```

Let's create a new table:

```
=> CREATE TABLE trans(
    id integer PRIMARY KEY,
    n integer
);

CREATE TABLE

=> INSERT INTO trans VALUES (1,10), (2,20), (3,30);

INSERT 0 3
```

To create transition tables for an operation, you have to specify their names in the trigger definition:

```
=> CREATE TRIGGER t_after_upd_trans
AFTER UPDATE ON trans -- one event per trigger
REFERENCING
    OLD TABLE AS old_table -- it's OK to specify only one table,
    NEW TABLE AS new_table -- there is no need to provide both
FOR EACH STATEMENT
EXECUTE FUNCTION transition();

CREATE TRIGGER
```

Let's check the result:

```
=> UPDATE trans SET n = n + 1 WHERE n <= 20;

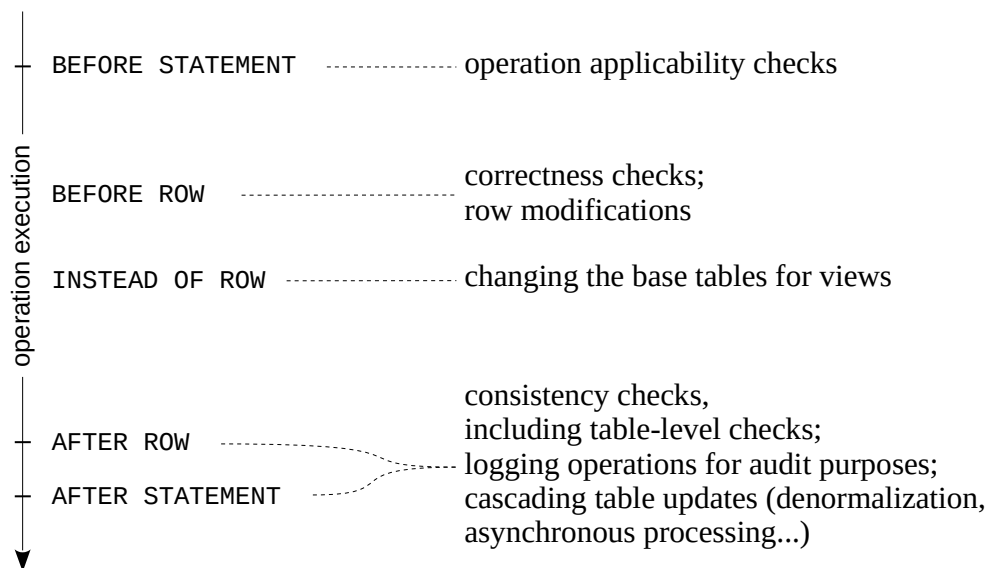
NOTICE: Old state:
NOTICE: (1,10)
NOTICE: (2,20)
NOTICE: New state:
NOTICE: (1,11)
NOTICE: (2,21)
UPDATE 2
```

Transition tables contain only those rows that have been affected by the operation.

In addition to updates, transition tables are also supported for INSERT and DELETE operations, although OLD TABLE is unavailable for inserts, while NEW TABLE is unavailable for deletes.

Since AFTER ROW triggers fire after the whole operation is completed, they can also use transition tables. But there is usually no point in it.

Possible Use Cases



How can we put this theory into practice?

BEFORE triggers can be used to check that the operation is valid and to raise errors if required.

BEFORE ROW triggers can be used to modify a row (for example, fill an empty field with the required value). It is convenient to use such triggers to avoid repeating the logic of filling out “technical” fields in each operation, as well as tweak the application behavior if its code cannot be modified.

INSTEAD OF ROW triggers are used to translate operations on views into the corresponding operations on the underlying base tables.

AFTER ROW and AFTER STATEMENT triggers can be useful for getting the exact state after the operation (BEFORE triggers may affect the result, so the state is not yet clear at this stage):

- to check consistency of the operation
- to perform audit operations, i.e., logging all changes in a separate storage
- to cascade changes to other tables (for example, to update denormalized data if the base tables have changed, or queue changes for subsequent processing outside of the current transaction)

If the operation affects multiple rows, it may be more efficient to use AFTER STATEMENT on transition tables instead of AFTER ROW as it can process changes in batches.

The code is called implicitly

the execution logic is hard to track

Visibility rules for volatile trigger functions

the result of BEFORE ROW and INSTEAD OF ROW triggers is visible

The order of calling triggers for one and the same event

triggers fire in the alphabetical order

Infinite looping can occur

a trigger can activate other triggers

Integrity constraints can be broken

for example, by excluding the rows that have to be deleted

Triggers should not be overused. As they fire implicitly, the logic of the application becomes obscure, thus making its maintenance hugely complicated. Attempts to use triggers for implementing complex logic are usually quite unfortunate.

In some cases, you can use generated columns instead of triggers (GENERATED ALWAYS AS . . . STORED). If applicable, this solution is sure to be more transparent and easier to implement.

There is a number of subtle points related to using triggers; we consciously skip their detailed discussion here:

- visibility rules of volatile functions in BEFORE ROW and INSTEAD OF ROW triggers (do not rely on the order of triggers when accessing a table)
- the order of calling several triggers on one and the same event (do not aggravate implicit firing of triggers by relying on their exact processing sequence)
- a possibility of infinite looping if cascade firing of triggers leads to another activation of the first trigger
- a risk of integrity constraint violation (for example, referential integrity can be compromised when skipping a row deleted by the ON DELETE CASCADE condition)

If you see that these subtleties are important for your application, you should seriously consider redesign.

Examples of Using Triggers

Example 1: saving the history of row updates.

Suppose we have a table that contains the current data. The task is to save the main table's history of all row updates into a separate table.

Historical table support could be delegated to the application, but chances are high that some part of the history won't be saved if an error occurs. That's why we are going to solve this problem using triggers.

The main table:

```
=> CREATE TABLE coins(  
    face_value numeric,  
    name text  
);
```

CREATE TABLE

To get the name of the historical table, we must extend the name of the main table with the "_history" suffix. We are going to clone the main table...

```
=> CREATE TABLE coins_history(LIKE coins);
```

CREATE TABLE

...and then add columns containing "valid from" and "valid until":

```
=> ALTER TABLE coins_history  
    ADD start_date timestamp,  
    ADD end_date timestamp;
```

ALTER TABLE

The first trigger function will be inserting a new historical row with an open validity interval:

```
=> CREATE OR REPLACE FUNCTION history_insert() RETURNS trigger  
AS $$  
BEGIN  
    EXECUTE format(  
        'INSERT INTO %I SELECT ($1).*, current_timestamp, NULL',  
        TG_TABLE_NAME || '_history'  
    ) USING NEW;  
  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

The second function will be closing the historical row's validity interval:

```
=> CREATE OR REPLACE FUNCTION history_delete() RETURNS trigger  
AS $$  
BEGIN  
    EXECUTE format(  
        'UPDATE %I SET end_date = current_timestamp WHERE face_value = $1 AND end_date IS NULL',  
        TG_TABLE_NAME || '_history'  
    ) USING OLD.face_value;  
  
    RETURN OLD;  
END;  
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

Now let's define triggers. Some important points to keep in mind:

- An update is treated as deletion followed by insertion; the order of triggers is important here (they fire in the alphabetical order).
- Current_timestamp returns the time of the transaction start, so if an update is performed, the start_date of one row will be the same as the end_date of another row.
- Using AFTER triggers allows avoiding issues with INSERT ... ON CONFLICT and potential conflicts with other triggers that may be defined on the main table.

```
=> CREATE TRIGGER coins_history_insert  
AFTER INSERT OR UPDATE ON coins  
FOR EACH ROW EXECUTE FUNCTION history_insert();
```

CREATE TRIGGER

```
=> CREATE TRIGGER coins_history_delete
AFTER UPDATE OR DELETE ON coins
FOR EACH ROW EXECUTE FUNCTION history_delete();
```

CREATE TRIGGER

Let's check our trigger implementation:

```
=> INSERT INTO coins VALUES (25, 'quarter'), (10, 'dime');
```

INSERT 0 2

```
=> UPDATE coins SET name = '10 cents' WHERE face_value = 10;
```

UPDATE 1

```
=> INSERT INTO coins VALUES (5, '5 cents');
```

INSERT 0 1

```
=> DELETE FROM coins WHERE face_value = 25;
```

DELETE 1

```
=> SELECT * FROM coins;
```

face_value	name
10	10 cents
5	5 cents

(2 rows)

The historical table stores the whole history of changes:

```
=> SELECT * FROM coins_history ORDER BY face_value, start_date;
```

face_value	name	start_date	end_date
5	5 cents	2021-10-19 17:03:29.640643	
10	dime	2021-10-19 17:03:26.467985	2021-10-19 17:03:28.604121
10	10 cents	2021-10-19 17:03:28.604121	
25	quarter	2021-10-19 17:03:26.467985	2021-10-19 17:03:31.672098

(4 rows)

And now we can use it to restore the state at any point in time (similar to the MVCC mechanism). For example, at the very beginning the table looked as follows:

```
=> \set d '2021-10-19 17:03:26.578844+03'
```

```
=> SELECT face_value, name
FROM coins_history
WHERE start_date <= :d AND (end_date IS NULL OR :d < end_date)
ORDER BY face_value;
```

face_value	name
10	dime
25	quarter

(2 rows)

Examples of Using Triggers

Example 2: an updatable view.

Suppose we have two tables: airports and flights:

```
=> CREATE TABLE airports(
    code char(3) PRIMARY KEY,
    name text NOT NULL
);
```

CREATE TABLE

```
=> INSERT INTO airports VALUES
('LHR', 'London. Heathrow'),
('CDG', 'Paris. Charles de Gaulle'),
('JFK', 'New York. John F. Kennedy');
```

INSERT 0 3


```
=> CREATE TABLE flights(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    airport_from char(3) NOT NULL REFERENCES airports(code),
    airport_to char(3) NOT NULL REFERENCES airports(code),
    UNIQUE (airport_from, airport_to)
);
```

CREATE TABLE

```
=> INSERT INTO flights(airport_from, airport_to) VALUES
    ('LHR', 'CDG');
```

INSERT 0 1

For convenience, we can define a view:

```
=> CREATE VIEW flights_v AS
SELECT id,
    (SELECT name
     FROM airports
     WHERE code = airport_from) airport_from,
    (SELECT name
     FROM airports
     WHERE code = airport_to) airport_to
FROM flights;
```

CREATE VIEW

```
=> SELECT * FROM flights_v;
```

```
id | airport_from | airport_to
---+-----+-----
 1 | London. Heathrow | Paris. Charles de Gaulle
(1 row)
```

But such a view does not support updates. For example, you won't be able to change the destination point using the following command:

```
=> UPDATE flights_v
SET airport_to = 'New York. John F. Kennedy'
WHERE id = 1;
```

ERROR: cannot update column "airport_to" of view "flights_v"
DETAIL: View columns that are not columns of their base relation are not updatable.

But we can define a trigger. A trigger function can look as follows (for brevity, we'll process only the destination airport, but it's not hard to add the departure airport as well):

```
=> CREATE OR REPLACE FUNCTION flights_v_update() RETURNS trigger
AS $$
DECLARE
    code_to char(3);
BEGIN
    BEGIN
        SELECT code INTO STRICT code_to
        FROM airports
        WHERE name = NEW.airport_to;
    EXCEPTION
        WHEN no_data_found THEN
            RAISE EXCEPTION 'Airport % is missing', NEW.airport_to;
    END;
    UPDATE flights
    SET airport_to = code_to
    WHERE id = OLD.id; -- ignore the id change
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

And the trigger itself will look like this:

```
=> CREATE TRIGGER flights_v_upd_trigger
INSTEAD OF UPDATE ON flights_v
FOR EACH ROW EXECUTE FUNCTION flights_v_update();
```

CREATE TRIGGER

Let's check the result:

```
=> UPDATE flights_v
SET airport_to = 'New York. John F. Kennedy'
WHERE id = 1;
```

UPDATE 1

```
=> SELECT * FROM flights_v;
```

id	airport_from	airport_to
1	London. Heathrow	New York. John F. Kennedy

(1 row)

An attempt to update the airport to the one missing from the table:

```
=> UPDATE flights_v  
SET airport_to = 'Amsterdam. Schiphol'  
WHERE id = 1;
```

ERROR: Airport Amsterdam. Schiphol is missing
CONTEXT: PL/pgSQL function flights_v_update() line 11 at RAISE

Event Triggers

An event trigger

is similar to a regular trigger defined on a table, but is a separate object

Trigger functions

convention: a function does not take any parameters,
returns a value of the `event_trigger` pseudotype

the call context is retrieved using special functions

Events

<code>DDL_COMMAND_START</code>	before command execution
<code>DDL_COMMAND_END</code>	after command execution
<code>TABLE_REWRITE</code>	before rewriting the table
<code>SQL_DROP</code>	after deleting objects

Event triggers are virtually the same as regular triggers, but instead of firing on DML operations, they fire on DDL operations (CREATE, ALTER, DROP, COMMENT, GRANT, REVOKE).

Such triggers are not an application development tool; they are mainly used for database administration purposes. We only mention them here to give you a complete picture, so we'll provide a very basic example.

<https://postgrespro.com/docs/postgresql/12/event-trigger-definition>

<https://postgrespro.com/docs/postgresql/12/event-trigger-matrix>

<https://postgrespro.com/docs/postgresql/12/sql-createeventtrigger>

<https://postgrespro.com/docs/postgresql/12/functions-event-triggers>

Event Triggers

Let's consider an example of a trigger for the `ddl_command_end` event, which corresponds to the end of a DDL operation.

Create a function that describes the context of the call:

```
=> CREATE OR REPLACE FUNCTION describe_ddl() RETURNS event_trigger
AS $$
DECLARE
    r record;
BEGIN
    -- For the ddl_command_end event, there is a special function for getting the context of the call
    FOR r IN SELECT * FROM pg_event_trigger_ddl_commands()
    LOOP
        RAISE NOTICE '%. type: %, OID: %, name: % ',
            r.command_tag, r.object_type, r.objid, r.object_identity;
    END LOOP;
    -- Event trigger functions do not have to return a value
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

The trigger itself will look as follows:

```
=> CREATE EVENT TRIGGER after_ddl
ON ddl_command_end EXECUTE FUNCTION describe_ddl();
```

CREATE EVENT TRIGGER

Let's create a new table:

```
=> CREATE TABLE t1(id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY);
```

```
NOTICE: CREATE SEQUENCE. type: sequence, OID: 16833, name: public.t1_id_seq
NOTICE: CREATE TABLE. type: table, OID: 16835, name: public.t1
NOTICE: CREATE INDEX. type: index, OID: 16838, name: public.t1_pkey
NOTICE: ALTER SEQUENCE. type: sequence, OID: 16833, name: public.t1_id_seq
CREATE TABLE
```

A table creation can lead to invocation of several DDL commands, that's why the `pg_event_trigger_ddl_commands` function returns a set of rows.

A trigger is a way to address a particular event

Using triggers, you can cancel an operation, modify its outcome, or perform additional actions

Triggers are executed as part of the main transaction; an error in a trigger aborts this transaction

Using AFTER ROW triggers and transition tables makes processing more expensive

Everything is good in moderation: complex logic is hard to debug because of implicit trigger execution



1. Create a trigger that handles updates of the `onhand_qty` field in the `catalog_v` view.
Check that the “Catalog” now allows ordering books.
2. Make sure that the following consistency requirement is met: the amount of available books cannot be negative (it is impossible to buy a book if it is not in stock).
Check your implementation carefully, keeping in mind that the application can be accessed by several users simultaneously.

Task 2. It may seem that it's enough to define the AFTER trigger on the `operations` table to calculate the `qty_change` sum. However, at the READ COMMITTED isolation level used in the “Bookstore” application, we will have to acquire an exclusive lock on this table: otherwise, the check may not function properly in some scenarios.

Here is a better approach: extend the `books` table with the `onhand_qty` column and create a trigger that will be modifying `onhand_qty` values when the `operations` table is changed (i.e., you should virtually perform data denormalization). You can now define the CHECK constraint on the `onhand_qty` field to ensure data consistency. The `onhand_qty()` function created earlier is no longer required.

You should pay special attention to setting the initial value, keeping in mind that the database system may be serving some users while we perform these changes.

Task 1. Using a Trigger to Update the Catalog

```
=> CREATE OR REPLACE FUNCTION update_catalog() RETURNS trigger
AS $$
BEGIN
    INSERT INTO operations(book_id, qty_change) VALUES
        (OLD.book_id, NEW.onhand_qty - coalesce(OLD.onhand_qty,0));
    RETURN NEW;
END;
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION

=> CREATE TRIGGER update_catalog_trigger
INSTEAD OF UPDATE ON catalog_v
FOR EACH ROW
EXECUTE FUNCTION update_catalog();

CREATE TRIGGER
```

Task 2. Checking the Quantity of Books

Let's extend the table with the column that will store the quantity of the books available. (For versions prior to 11, it is important to take into account that the DEFAULT clause will cause overwriting of all table rows while holding the lock.)

```
=> ALTER TABLE books ADD COLUMN onhand_qty integer;
```

```
ALTER TABLE
```

The trigger function for the AFTER trigger, which is fired on insertion to update the quantity of available books (we assume that the onhand_qty field cannot be empty):

```
=> CREATE OR REPLACE FUNCTION update_onhand_qty() RETURNS trigger
AS $$
BEGIN
    UPDATE books
    SET onhand_qty = onhand_qty + NEW.qty_change
    WHERE book_id = NEW.book_id;
    RETURN NULL;
END;
$$ VOLATILE LANGUAGE plpgsql;

CREATE FUNCTION
```

The remaining operations are performed within a single transaction.

```
=> BEGIN;
```

```
BEGIN
```

Locking the table for the duration of the transaction:

```
=> LOCK TABLE operations;
```

```
LOCK TABLE
```

Providing the initial value:

```
=> UPDATE books b
SET onhand_qty = (
    SELECT coalesce(sum(qty_change),0)
    FROM operations o
    WHERE o.book_id = b.book_id
);

UPDATE 7
```

Defining constraints now that the field is non-empty:

```
=> ALTER TABLE books ALTER COLUMN onhand_qty SET DEFAULT 0;
```

```
ALTER TABLE
```

```
=> ALTER TABLE books ALTER COLUMN onhand_qty SET NOT NULL;
```

```
ALTER TABLE
```

```
=> ALTER TABLE books ADD CHECK(onhand_qty >= 0);
```

```
ALTER TABLE
```

Creating a trigger:

```
=> CREATE TRIGGER update_onhand_qty_trigger
AFTER INSERT ON operations
FOR EACH ROW
EXECUTE FUNCTION update_onhand_qty();
```

CREATE TRIGGER

Done.

```
=> COMMIT;
```

COMMIT

Now the books.onhand_qty column is being updated, but the catalog_v view still calls a function to calculate the number of books. Although the syntax used for function access in the initial query is the same as that for the field access, the query has been stored in a different form:

```
=> \d+ catalog_v
```

View "bookstore.catalog_v"						
Column	Type	Collation	Nullable	Default	Storage	Description
book_id	integer				plain	
title	text				extended	
onhand_qty	integer				plain	
display_name	text				extended	
authors	text				extended	

View definition:

```
SELECT b.book_id,
       b.title,
       onhand_qty(b.*) AS onhand_qty,
       book_name(b.book_id, b.title) AS display_name,
       authors(b.*) AS authors
FROM books b
ORDER BY (book_name(b.book_id, b.title));
```

Triggers:

```
update_catalog_trigger INSTEAD OF UPDATE ON catalog_v FOR EACH ROW EXECUTE FUNCTION update_catalog()
```

Let's replace the view:

```
=> CREATE OR REPLACE VIEW catalog_v AS
SELECT b.book_id,
       b.title,
       b.onhand_qty,
       book_name(b.book_id, b.title) AS display_name,
       b.authors
FROM books b
ORDER BY display_name;
```

CREATE VIEW

Now the function can be deleted.

```
=> DROP FUNCTION onhand_qty(books);
```

DROP FUNCTION

Let's run a small check:

```
=> SELECT * FROM catalog_v WHERE book_id = 1 \gx
```

```
-[ RECORD 1 ]+-----
book_id      | 1
title        | The Tale of Tsar Saltan
onhand_qty   | 19
display_name | The Tale of Tsar Saltan. Alexander S. Pushkin
authors      | Alexander Sergeyevich Pushkin
```

```
=> INSERT INTO operations(book_id, qty_change) VALUES (1,+10);
```

INSERT 0 1

```
=> SELECT * FROM catalog_v WHERE book_id = 1 \gx
```

```
-[ RECORD 1 ]+-----
book_id      | 1
title        | The Tale of Tsar Saltan
onhand_qty   | 29
display_name | The Tale of Tsar Saltan. Alexander S. Pushkin
authors      | Alexander Sergeyevich Pushkin
```


Incorrect operations are aborted:

```
=> INSERT INTO operations(book_id, qty_change) VALUES (1,-100);
```

ERROR: new row for relation "books" violates check constraint "books_onhand_qty_check"

DETAIL: Failing row contains (1, The Tale of Tsar Saltan, -71).

CONTEXT: SQL statement "UPDATE books

SET onhand_qty = onhand_qty + NEW.qty_change

WHERE book_id = NEW.book_id"

PL/pgSQL function update_onhand_qty() line 3 at SQL statement

1. Create a trigger that increments the counter (the version field) by 1 each time a row is updated. If a new row is inserted, the corresponding counter must be set to 1.
Check the implementation.
2. There are two tables: orders (to store orders) and lines (to keep order lines).
It is required to perform denormalization: automatically update the total cost of the order in the orders table once the corresponding lines have changed.
Create the required triggers using transition tables to minimize the number of update operations.

Task 2. Use the following commands to create tables:

```
CREATE TABLE orders (  
    id int PRIMARY KEY,  
    total_amount numeric(20,2) NOT NULL DEFAULT 0  
);  
  
CREATE TABLE lines (  
    id int PRIMARY KEY GENERATED ALWAYS AS IDENTITY,  
    order_id int NOT NULL REFERENCES orders(id),  
    amount numeric(20,2) NOT NULL  
);
```

The `orders.total_amount` column must be calculated automatically as the sum of `lines.amount` values for all rows related to the corresponding order.

Task 1. Version Count

```
=> CREATE DATABASE plpgsql_triggers;
```

CREATE DATABASE

```
=> \c plpgsql_triggers
```

You are now connected to database "plpgsql_triggers" as user "student".

A table:

```
=> CREATE TABLE t(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    s text,
    version integer
);
```

CREATE TABLE

A trigger function:

```
=> CREATE OR REPLACE FUNCTION inc_version() RETURNS trigger
AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.version := 1;
    ELSE
        NEW.version := OLD.version + 1;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

A trigger:

```
=> CREATE TRIGGER t_inc_version
BEFORE INSERT OR UPDATE ON t
FOR EACH ROW EXECUTE FUNCTION inc_version();
```

CREATE TRIGGER

The result:

```
=> INSERT INTO t(s) VALUES ('One');
```

INSERT 0 1

```
=> SELECT * FROM t;
```

id	s	version
1	One	1

(1 row)

An explicit specification of version is ignored:

```
=> INSERT INTO t(s,version) VALUES ('Two',42);
```

INSERT 0 1

```
=> SELECT * FROM t;
```

id	s	version
1	One	1
2	Two	1

(2 rows)

Let's perform an update:

```
=> UPDATE t SET s = lower(s) WHERE id = 1;
```

UPDATE 1

```
=> SELECT * FROM t;
```

id	s	version
2	Two	1
1	one	2

(2 rows)

The explicit specification is still ignored:

```
=> UPDATE t SET s = lower(s), version = 42 WHERE id = 2;
```

UPDATE 1

```
=> SELECT * FROM t;
```

id	s	version
1	one	2
2	two	2

(2 rows)

Task 2. Calculating the Total Cost of Orders Automatically

Let's create some tables with a simplified structure that is sufficient for demo purposes:

```
=> CREATE TABLE orders (
    id integer PRIMARY KEY,
    total_amount numeric(20,2) NOT NULL DEFAULT 0
);
```

CREATE TABLE

```
=> CREATE TABLE lines (
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    order_id integer NOT NULL REFERENCES orders(id),
    amount numeric(20,2) NOT NULL
);
```

CREATE TABLE

A trigger function and a trigger to process insertions:

```
=> CREATE FUNCTION total_amount_ins() RETURNS trigger
AS $$
BEGIN
    WITH l(order_id, total_amount) AS (
        SELECT order_id, sum(amount)
        FROM new_table
        GROUP BY order_id
    )
    UPDATE orders o
    SET total_amount = o.total_amount + l.total_amount
    FROM l
    WHERE o.id = l.order_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

CREATE FUNCTION

The FROM clause in the UPDATE command allows us to join orders with a subquery on the transition table and to use subquery columns when calculating the total_amount value.

```
=> CREATE TRIGGER lines_total_amount_ins
AFTER INSERT ON lines
REFERENCING
    NEW TABLE AS new_table
FOR EACH STATEMENT
EXECUTE FUNCTION total_amount_ins();
```

CREATE TRIGGER

A trigger function and a trigger to process updates:

```

=> CREATE FUNCTION total_amount_upd() RETURNS trigger
AS $$
BEGIN
    WITH l_tmp(order_id, amount) AS (
        SELECT order_id, amount FROM new_table
        UNION ALL
        SELECT order_id, -amount FROM old_table
    ), l(order_id, total_amount) AS (
        SELECT order_id, sum(amount)
        FROM l_tmp
        GROUP BY order_id
        HAVING sum(amount) <> 0
    )
    UPDATE orders o
    SET total_amount = o.total_amount + l.total_amount
    FROM l
    WHERE o.id = l.order_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION

```

The HAVING condition allows skipping changes that do not affect the total cost of the order.

```

=> CREATE TRIGGER lines_total_amount_upd
AFTER UPDATE ON lines
REFERENCING
    OLD TABLE AS old_table
    NEW TABLE AS new_table
FOR EACH STATEMENT
EXECUTE FUNCTION total_amount_upd();

CREATE TRIGGER

```

A trigger function and a trigger to process deletion:

```

=> CREATE FUNCTION total_amount_del() RETURNS trigger
AS $$
BEGIN
    WITH l(order_id, total_amount) AS (
        SELECT order_id, -sum(amount)
        FROM old_table
        GROUP BY order_id
    )
    UPDATE orders o
    SET total_amount = o.total_amount + l.total_amount
    FROM l
    WHERE o.id = l.order_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION

=> CREATE TRIGGER lines_total_amount_del
AFTER DELETE ON lines
REFERENCING
    OLD TABLE AS old_table
FOR EACH STATEMENT
EXECUTE FUNCTION total_amount_del();

CREATE TRIGGER

```

We have not addressed the TRUNCATE operator yet. But a trigger for this operator cannot use transition tables. However, we know that there will be no lines left after TRUNCATE execution, so we can zero out the cost of all orders.

```

=> CREATE FUNCTION total_amount_truncate() RETURNS trigger
AS $$
BEGIN
    UPDATE orders SET total_amount = 0;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION

=> CREATE TRIGGER lines_total_amount_truncate
AFTER TRUNCATE ON lines
FOR EACH STATEMENT
EXECUTE FUNCTION total_amount_truncate();

CREATE TRIGGER

```

Additionally, we have to forbid changing the total_amount value manually, but this problem is solved without using triggers.

Let's check the result.

Add two new orders without any lines:

```
=> INSERT INTO orders VALUES (1), (2);
```

INSERT 0 2

```
=> SELECT * FROM orders ORDER BY id;
```

id	total_amount
1	0.00
2	0.00

(2 rows)

Add lines for these orders:

```
=> INSERT INTO lines (order_id, amount) VALUES
(1,100), (1,100), (2,500), (2,500);
```

INSERT 0 4

```
=> SELECT * FROM lines;
```

id	order_id	amount
1	1	100.00
2	1	100.00
3	2	500.00
4	2	500.00

(4 rows)

```
=> SELECT * FROM orders ORDER BY id;
```

id	total_amount
1	200.00
2	1000.00

(2 rows)

Double the sum of all lines of all orders:

```
=> UPDATE lines SET amount = amount * 2;
```

UPDATE 4

```
=> SELECT * FROM orders ORDER BY id;
```

id	total_amount
1	400.00
2	2000.00

(2 rows)

Now delete one line of the first order:

```
=> DELETE FROM lines WHERE id = 1;
```

DELETE 1

```
=> SELECT * FROM orders ORDER BY id;
```

id	total_amount
1	200.00
2	2000.00

(2 rows)

Truncate the lines table:

```
=> TRUNCATE lines;
```

TRUNCATE TABLE

```
=> SELECT * FROM orders ORDER BY id;
```

id	total_amount
1	0.00
2	0.00

(2 rows)