Triggers - Friends To Handle With Care

Charles Clavadetscher

Swiss PostgreSQL Users Group

pgDay Paris, 15.03.2018, Paris, France

Outline

In Short About Me

- Senior DB Engineer at KOF ETH Zurich
  - KOF is the Center of Economic Research of the
  - ETHZ the Swiss Institute of Technology in Zurich, Switzerland
  - Independent economic research on business cycle tendencies for almost all sectors
  - Maintenance of all databases at KOF: PostgreSQL, Oracle, MySQL and MSSQL Server. Focus on migrating to PostgreSQL
  - Support in business process re-engineering
- Co-founder and treasurer of the SwissPUG, the Swiss PostgreSQL Users Group
- Member of the board of the Swiss PGDay

Outline

Introduction

What is a trigger ?

From Wikipedia (https://en.wikipedia.org/wiki/Database_trigger): A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database.

Introduction

What is a trigger ?

Be aware: If you ever created a constraint (index, foreign key, check, etc.) you have been using triggers.

This presentation is about triggers that are specified **explicitly** by a user.

Triggers can be speficied on

- Tables
- Foreign tables
- Views

Introduction

Create a trigger

Basic workflow

- Each time that one of a list of commands tries to change the data of a specified table
  - When a whole row is inserted or deleted
  - When specific or all columns are changed
  - When other characteristics are met
- Execute a function
  - Do whatever needs to be done
  - Inform the calling process what to do with the data...
  - ... Returning the data to be acted upon or null or throwing an exception
- Before or after the table content is changed
- For each row that should be changed or only once for the whole statement

Introduction

A Trigger Function

```
CREATE OR REPLACE FUNCTION fname()
RETURNS TRIGGER
AS $$
BEGIN
[...]
END;
$$ LANGUAGE plpgsql;
```

- The function has no parameters.
- It returns the type `trigger`.
- Receives its input through special variables `NEW` and `OLD`.
- The languge for the function's implementation can be any of the many procedural languages available as extensions in PostgreSQL.

Introduction

A Trigger Function: Variables

- `NEW`: Data type RECORD ; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is unassigned in statement-level triggers and for DELETE operations.
- `OLD`: Data type RECORD ; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is unassigned in statement-level triggers and for INSERT operations.
- `TG_OP`: Data type text ; a string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired.
- `TG_NAME`: Data type name ; variable that contains the name of the trigger actually fired. Could be helpful for reporting.
- `TG_TABLE_NAME`: Data type name ; the name of the table that caused the trigger invocation.
- `TG_TABLE_SCHEMA`: Data type name ; the name of the schema of the table that caused the trigger invocation.

Introduction

Create a trigger

Set up the basic flow

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table_name
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Introduction

Create a trigger

More settings

```
db=> \h CREATE TRIGGER

CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Introduction

New In PostgreSQL 10: Transition tables

Keep track of change summaries.

```
CREATE OR REPLACE FUNCTION public.rows_modified()
RETURNS TRIGGER
AS $$
DECLARE
  v_msg TEXT;
  v_tot INTEGER;
  v_avg_old NUMERIC(6,2);
  v_avg_new NUMERIC(6,2);
BEGIN
  SELECT count(1),
         avg(oldtab.price),
         avg(newtab.price)
  INTO v_tot, v_avg_old, v_avg_new
  FROM newtab, oldtab
  WHERE newtab.book_id = oldtab.book_id;

  v_msg := 'Modified ' || v_tot || ' rows. Old average price: ' ||
           v_avg_old || ', new average price: ' || v_avg_new;

  INSERT INTO public.books_log (log_f, log_msg)
  VALUES ('public.rows_modified()',v_msg);
  RAISE NOTICE '%', v_msg;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Introduction

New In PostgreSQL 10: Transition tables

Request transition tables for the trigger.

```
CREATE TRIGGER rows_modified
AFTER UPDATE ON public.books
REFERENCING NEW TABLE newtab OLD TABLE oldtab
FOR EACH STATEMENT
EXECUTE PROCEDURE public.rows_modified();

db=# SELECT count(1) FROM public.books WHERE price < 1.0;
-[ RECORD 1 ]
count | 4

db=# UPDATE public.books set price = price * 2 where price < 1.0 ;

NOTICE:  Price of Hans Peter Roth Orte des Grauens in der Schweiz changed from CHF   [...]
NOTICE:  Price of Betty Bossi Das grosse Betty Bossi Kochbuch changed from CHF 0.25  [...]
NOTICE:  Price of Werner König DTV Atlas der deutschen Sprache changed from CHF 0.41 [...]
NOTICE:  Price of Otto Hostettler Darknet changed from CHF 0.55 to CHF 1.10
NOTICE:  Modified 4 rows. Old average price: 0.47, new average price: 0.93
UPDATE 4
```

Also available for row level triggers.

Outline

Triggers Security

Who can create a trigger ?

A trigger can be specified by users having the trigger privilege on the object for which the trigger is being created.

Avoid granting this privilege to users unless you know who you are granting it to. Triggers perform mostly silently and users may misuse the feature maliciously or even create obscure performance problems.

Triggers Security

Who can execute a trigger ?

A trigger specifies a function that is not called directly. The only way to invoke the function is through an event. A user who has privileges to modify data in a table will be able to execute the function defined for a trigger without needing an explicit EXECUTE privilege.

BUT restrictions that apply to the execution steps within the body of the trigger function follow the same rule as for functions in general. In particular the CURRENT_USER must have any required privilege on objects touched by the trigger function.

Triggers Security

Who can execute a trigger ?

As privileged user.

```
db=> GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE public.books TO genericuser;
```

As genericuser.

```
db=> SELECT SESSION_USER, CURRENT_USER;
 session_user | current_user
--------------+--------------
 genericuser  | genericuser

db=> UPDATE public.books SET price = 11.00 WHERE book_id = 1;
ERROR:  permission denied for relation books_log
CONTEXT:  SQL statement "INSERT INTO public.books_log (log_f, log_msg)
  VALUES ('public.price_changed()',v_msg)"
PL/pgSQL function price_changed() line 6 at SQL statement
```

You may choose to grant INSERT to genericuser or to declare the trigger function as SECURITY DEFINER. Which is better depends on your policies.

Outline

Manage Triggers

Modify a trigger

```
db=> \h ALTER TRIGGER
Command:     ALTER TRIGGER
Description: change the definition of a trigger
Syntax:
ALTER TRIGGER name ON table_name RENAME TO new_name
ALTER TRIGGER name ON table_name DEPENDS ON EXTENSION extension_name

db=> \h DROP TRIGGER
Command:     DROP TRIGGER
Description: remove a trigger
Syntax:
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

Real changes to trigger behaviour are changes in the trigger functions.

Manage Triggers

Disable a trigger

```
db=> ALTER TABLE public.books DISABLE TRIGGER price_changed ;
db=> \d public.books
[...]
Triggers:
    rows_modified AFTER UPDATE ON books REFERENCING OLD TABLE AS oldtab [...]
Disabled user triggers:
    price_changed AFTER UPDATE ON books FOR EACH ROW [...]

db=> ALTER TABLE public.books ENABLE TRIGGER price_changed ;
db=> \d public.books
[...]
Triggers:
    price_changed AFTER UPDATE ON books FOR EACH ROW [...]
    rows_modified AFTER UPDATE ON books REFERENCING OLD TABLE AS oldtab [...]
```

Manage Triggers

Find triggers: Which tables have triggers ?

```
db=> SELECT n.nspname || '.' || c.relname AS table_name
     FROM pg_catalog.pg_namespace n,
          pg_catalog.pg_class c
     WHERE c.relhastriggers
     AND n.nspname !~ '^pg_'
     AND n.nspname <> 'information_schema'
     ORDER BY table_name;

    table_name
------------------
 public.books
```

```
db=> SELECT event_object_schema || '.' || event_object_table AS tablename,
            trigger_name FROM information_schema.triggers
     ORDER BY tablename,
            trigger_name ;

  tablename   | trigger_name
--------------+---------------
 public.books | price_changed
 public.books | rows_modified
```

Manage Triggers

Find triggers: Which tables share a trigger ?

Careful: trigger name <> trigger function. The same trigger name may have different functions defined for different tables.

```
db=> CREATE TRIGGER price_changed AFTER UPDATE ON TABLE books_bak
     FOR EACH STATEMENT EXECUTE PROCEDURE rows_modified();

db=> SELECT event_object_schema || '.' || event_object_table as tablename,
            trigger_name,
            action_statement
     FROM information_schema.triggers
     WHERE trigger_name = 'price_changed' ;

    tablename     | trigger_name  |          action_statement
------------------+---------------+-----------------------------------
 public.books     | price_changed | EXECUTE PROCEDURE price_changed()
 public.books_bak | price_changed | EXECUTE PROCEDURE rows_modified()
```

Manage Triggers

Find triggers: Which tables share a trigger ?

```
db=> WITH tf AS (
       SELECT tgfoid     <-- Get the function OID
       FROM pg_catalog.pg_trigger
       WHERE tgname = 'rows_modified'          <-- The name of the trigger
       AND tgrelid = 'public.books'::regclass  <-- On this table
     ) -- Find all tables using the same function in a trigger
     SELECT n.nspname || '.' || c.relname AS table_name,
            t.tgname AS trigger_name,
            p.proname AS func_name
     FROM pg_catalog.pg_namespace n,
          pg_catalog.pg_class c,
          pg_catalog.pg_trigger t,
          pg_catalog.pg_proc p,
          tf
     WHERE t.tgfoid = tf.tgfoid
     AND n.oid = c.relnamespace
     AND t.tgrelid = c.oid
     AND p.oid = tf.tgfoid;

    table_name     | trigger_name   |  func_name
-------------------+----------------+---------------
 public.books      | rows_modified  | rows_modified
 public.books_bak  | price_changed  | rows_modified
```

Manage Triggers

Find triggers: Which triggers has a table ?

Shortcut for a single table if you use **the best** PostgreSQL client.

```
db=> \d public.books
                        Table "public.books"
    Column     |            Type             | Collation | Nullable |
---------------+-----------------------------+-----------+----------+
 book_id       | bigint                      |           | not null |
[...]
Indexes:
    "books_pkey" PRIMARY KEY, btree (book_id)
Triggers:
    price_changed AFTER UPDATE ON books FOR EACH ROW
      WHEN (old.price IS DISTINCT FROM new.price OR
                    old.currency IS DISTINCT FROM new.currency)
      EXECUTE PROCEDURE price_changed()
    rows_modified AFTER UPDATE ON books
      REFERENCING OLD TABLE AS oldtab NEW TABLE AS newtab
      FOR EACH STATEMENT
      EXECUTE PROCEDURE rows_modified()
```

Manage Triggers

What does a trigger do ?

```
db=> SELECT pg_catalog.pg_get_functiondef((SELECT tgfoid
                                           FROM pg_catalog.pg_trigger
                                           WHERE tgname = 'price_changed'));

                              pg_get_functiondef
-------------------------------------------------------------------------------
 CREATE OR REPLACE FUNCTION public.price_changed()
  RETURNS trigger
  LANGUAGE plpgsql
 AS $function$
 DECLARE
   v_msg TEXT;
 BEGIN
   v_msg := 'Price of '||NEW.author||' '||NEW.title||' changed from '||OLD.currency||
                      ' '||OLD.price||' to '||NEW.currency||' '||NEW.price;
   INSERT INTO public.books_log (log_f, log_msg)
   VALUES ('public.price_changed()',v_msg);
   RAISE NOTICE '%', v_msg;
   RETURN NULL;
 END;
 $function$
```

Outline

Use Cases And Pitfalls

Most typical Use Cases

- Keeping track of changes.
  - History.
  - Audit.
- Complex checks before modifications in the database.
- Enforce complex business rules.
- Create additional related entries.
- Protect data.

Use Cases And Pitfalls

Order of triggers

```
SELECT trigger_name,
       event_object_schema AS schemaname,
       event_object_table AS tablename,
       action_timing AS timing,
       event_manipulation AS events,
       replace(action_statement, 'EXECUTE PROCEDURE ','') AS function
FROM information_schema.triggers
ORDER BY action_timing DESC,
       trigger_name ;
```

```
 trigger_name   | schemaname | tablename | timing | events |     function
----------------+------------+-----------+--------+--------+-----------------
 price_changed | public     | books     | AFTER  | UPDATE | price_changed()
 rows_modified | public     | books     | AFTER  | UPDATE | rows_modified()
```

Use Cases And Pitfalls

Histories And DDL changes: Create a history table and trigger

```
CREATE TABLE public.books_history AS
  SELECT *, NULL::TEXT AS change_op, NULL::TEXT AS change_user,
            NULL::TIMESTAMPTZ AS change_ts
  FROM public.books LIMIT 0;

CREATE OR REPLACE FUNCTION public.books_history()
RETURNS TRIGGER
AS $$
BEGIN
  INSERT INTO public.books_history
  SELECT NEW.*, TG_OP, SESSION_USER, clock_timestamp();
  CASE WHEN TG_OP = 'DELETE' THEN RETURN OLD;
       ELSE RETURN NEW;
  END CASE;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER books_history
AFTER INSERT OR UPDATE OR DELETE
ON public.books
FOR EACH ROW EXECUTE PROCEDURE public.books_history();
```

Use Cases And Pitfalls

Histories And DDL changes: What happens if your table structure changes ?

```
db=> UPDATE public.books SET title = 'La divina commedia', price = 10.66 WHERE book_id = 1 ;
[...]
UPDATE 1

db=> SELECT * FROM public.books_history;
-[ RECORD 1 ]-+----------------------------
book_id       | 1
author        | Dante Alighieri
title         | La divina commedia
[...]
change_op     | UPDATE
change_user   | charles
change_ts     | 2018-01-23 09:39:56.437427+01
```

And now...

```
ALTER TABLE public.books ALTER COLUMN last_modified TYPE TIMESTAMP;

db=> UPDATE public.books SET title = 'La divina commedia', price = 10.66 WHERE book_id = 1 ;

ERROR:  attribute 6 has wrong type <-- What is the problem?
CONTEXT:  SQL statement "INSERT INTO public.books_history -+
  SELECT NEW.*, TG_OP, SESSION_USER, clock_timestamp()"    |-- Where did it happen?
PL/pgSQL function books_history() line 3 at SQL statement -+
```

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

Triggers can be an efficient way to protect data against unintended changes. In a real use case we had the following situation.

- A table recording answers to time series surveys.
- The table only inserts when new answers come into the system.
- Answers can come from a web frontend form or from scanned paper forms. **In latter case data is inserted into a flat table with a trigger that reformats data and makes the inserts into the time series survey data target table**.

The World was kind and simple till new requirements came up...

- It must be possible to find the earliest invitation to a survey.
- It must be possible to track the basic answering discipline of respondents.
- It must be possible to analyze every aspect of non respondents.
- This information must be retrieved very quickly and displayed in a Web GUI.

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

In short: We need to create entries for invitations, e.g. empty form entries in the time series survey table and modify the interface for paper forms data input (the trigger) to allow updates, instead of or additionally to inserts.

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

```
db=> \d survey_data
            Table "public.survey_data"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 sp_id  | integer |           | not null |
 year   | integer |           | not null |
 month  | integer |           | not null |
 roa    | integer |           |          |
 v1     | integer |           |          |
 v2     | integer |           |          |
Indexes:
    "sd_pkey" PRIMARY KEY, btree (sp_id, year, month)

db=> SELECT * FROM public.survey_data;
 sp_id  | year | month | roa | v1 | v2
--------+------+-------+-----+----+----
 119903 | 2017 |     9 |   1 |  0 |  0
 117278 | 2018 |     1 |   2 |  0 |  0
 115709 | 2018 |     1 |   2 | -1 |  0
 117147 | 2018 |     1 |   2 | -1 |  0
 115581 | 2018 |     1 |   2 |  1 |  0
 115496 | 2018 |     1 |   1 |  1 |  0
[...]
```

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

The nice brave World of before...

```
db=> SELECT year, month, count(*) AS total_forms,
            avg(v1)::NUMERIC(6,3) AS v1_avg,
            avg(v2)::NUMERIC(6,3) AS v2_avg
     FROM public.survey_data
     GROUP BY year, month ORDER BY year, month;

 year | month | total_forms | v1_avg | v2_avg
------+-------+-------------+--------+--------
 2017 |     7 |         183 | -0.188 |  0.068
 2017 |     8 |         180 | -0.217 |  0.119
 2017 |     9 |         184 | -0.233 |  0.098
 2017 |    10 |         180 | -0.196 |  0.088
 2017 |    11 |         189 | -0.257 |  0.061
 2017 |    12 |         184 | -0.137 |  0.096
(6 rows)
```

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

Now we should find answers to the new requirements. Before including empty records:

```
db=> SELECT year, month, count(*) AS total_forms,
            count(*) FILTER (WHERE roa IS NOT NULL) AS answered_forms,
            avg(v1)::NUMERIC(6,3) AS v1_avg,
            avg(v2)::NUMERIC(6,3) AS v2_avg
     FROM public.survey_data
     GROUP BY year, month ORDER BY year, month;

 year | month | total_forms | answered_forms | v1_avg | v2_avg
------+-------+-------------+----------------+--------+--------
[...]
 2017 |    11 |         189 |            189 | -0.257 |  0.061
 2017 |    12 |         184 |            184 | -0.137 |  0.096
 2018 |     1 |         184 |            184 | -0.267 |  0.069
```

After change:

```
 year | month | total_forms | answered_forms | v1_avg | v2_avg
------+-------+-------------+----------------+--------+--------
[...]
 2017 |    11 |         256 |            189 | -0.257 |  0.061
 2017 |    12 |         252 |            184 | -0.137 |  0.096
 2018 |     1 |         265 |            184 | -0.267 |  0.069
```

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

Old Trigger:

```
BEGIN
  INSERT INTO public.survey_data
  VALUES (NEW.sp_id, NEW.year, NEW.month, NEW.roa, NEW.v1, NEW.v2);
  RETURN NEW;
END;

CREATE TRIGGER trg_add_paper_forms
BEFORE INSERT ON public.survey_data_paper
FOR EACH ROW EXECUTE PROCEDURE public.trg_add_paper_forms();
```

New Trigger:

```
CREATE OR REPLACE FUNCTION public.trg_add_paper_forms()
RETURNS TRIGGER
AS $$
BEGIN
  UPDATE public.survey_data
  SET roa = NEW.roa,
      v1 = NEW.v1,
      v2 = NEW.v2
  WHERE sp_id = NEW.sp_id;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

Nice, but after loading paper data for 2018 January we found that...

```
 year | month | total_forms | answered_forms | v1_avg | v2_avg
------+-------+-------------+----------------+--------+--------
[...]
 2017 |    11 |         256 |            178 | -0.259 |  0.078
 2017 |    12 |         252 |            179 | -0.263 |  0.071
 2018 |     1 |         265 |            184 | -0.267 |  0.069
(7 rows)
```

... the values in the past changed. They were:

```
 year | month | total_forms | answered_forms | v1_avg | v2_avg
------+-------+-------------+----------------+--------+--------
[...]
 2017 |    11 |         256 |            189 | -0.257 |  0.061
 2017 |    12 |         252 |            184 | -0.137 |  0.096
 2018 |     1 |         265 |            184 | -0.267 |  0.069
```

What caused the change ?

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

After restoring the data from the past we added a data protection trigger.

```
CREATE OR REPLACE FUNCTION public.protect_past_survey_data()
RETURNS TRIGGER
AS $$
BEGIN
  -- If data that is being changed is not in the current month.
  IF make_date(NEW.year, NEW.month, 1) < date_trunc('month', CURRENT_DATE) THEN
    -- Stop execution and inform user.
    RAISE EXCEPTION 'It is not allowed to change past data (%)', NEW;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_protect_survey_data
BEFORE INSERT OR UPDATE ON public.survey_data
FOR EACH ROW EXECUTE PROCEDURE public.protect_past_survey_data();
```

Use Cases And Pitfalls

Errors In Functions: How A Trigger Solved A Trigger Problem

Trying to update paper form data for 2018 month 1 led to this error:

```
ERROR:  It is not allowed to change past data ((117278,2017,9,2,0,0))
CONTEXT:  PL/pgSQL function protect_past_survey_data() line 4 at RAISE
SQL statement "UPDATE public.survey_data
  SET roa = NEW.roa,
      v1 = NEW.v1,
      v2 = NEW.v2
  WHERE sp_id = NEW.sp_id"
PL/pgSQL function trg_add_paper_forms() line 3 at SQL statement
```

Now it became clear where the error was and solve it:

```
BEGIN
  UPDATE public.survey_data
  SET roa = NEW.roa,
      v1 = NEW.v1,
      v2 = NEW.v2
  WHERE sp_id = NEW.sp_id
  AND year = NEW.year
  AND month = NEW.month;
  RETURN NEW;
END;
```

Outline

Triggers And Other Stories

Errors In Functions: How A Trigger Solved A Trigger Problem

Ever heard ?
Triggers are useless for auditing because the audit data is in the same database that is being audited.
Well, heard of **foreign data wrappers** ?

Triggers And Other Stories

Manage Histories and Audits On A Separate Database With FDW

```
CREATE EXTENSION postgres_fdw;
CREATE SERVER dbremote FOREIGN DATA WRAPPER postgres_fdw
  OPTIONS (host 'localhost', dbname 'dbremote', port '5432');
CREATE USER MAPPING FOR charles SERVER dbremote
  OPTIONS (user 'remoteuser', password '...');

CREATE FOREIGN TABLE public.books_history_remote (
  book_id BIGINT,
  author TEXT,
  title TEXT,
  currency TEXT,
  price NUMERIC(6,2),
  last_modified TIMESTAMPTZ,
  visible BOOLEAN,
  change_op TEXT,
  change_user TEXT,
  change_ts TIMESTAMPTZ
)
SERVER dbremote
OPTIONS (
    schema_name 'public',
    table_name 'books_history',
    updatable 'true'
);
```

Triggers And Other Stories

Manage Histories and Audits On A Separate Database

```
CREATE OR REPLACE FUNCTION public.books_history()
RETURNS TRIGGER
AS $$
BEGIN
  INSERT INTO public.books_history_remote
  SELECT NEW.*, TG_OP, SESSION_USER, clock_timestamp();
  CASE WHEN TG_OP = 'DELETE' THEN RETURN OLD;
       ELSE RETURN NEW;
  END CASE;
END;
$$ LANGUAGE plpgsql;

db=> UPDATE public.books SET price = 11.10 WHERE book_id = 1 ;
UPDATE 1
db=> SELECT * FROM books_history_remote;
-[ RECORD 1 ]-+----------------------------
book_id       | 1
author        | Dante Alighieri
title         | La divina commedia
currency      | CHF
price         | 11.10
last_modified | 2018-01-08 14:06:35.197023+01
visible       | t
change_op     | UPDATE
change_user   | charles
change_ts     | 2018-01-23 11:27:21.768927+01
```

Outline

Recommendations

The usage of triggers is easy, and as easy as that is, easy it is to do damage. But with some simple guidelines you should be able to avoid problems.

- Triggers are a powerful mechanism for automating processes in a database.
- If your application interface is made of functions, use triggers only for requirements not covered by them.
- Take special care and make extensive tests before setting a trigger into production.
- Test your trigger function with real data, not just a few test data.
- Make a backup of your data before launching a trigger into production.
- If you have more than one trigger on a table, make sure the order of them does what you intend it to do.
- Let your triggers tell you what they are doing and when.
- **Document the intended behaviour of your triggers** (e.g. with COMMENT ON TRIGGER).

Resources

These slides

- http://www.artesano.ch/documents/04-publications/triggers_pdfa.pdf

Official PostgreSQL documentation

- Description (Chapter 38): https://www.postgresql.org/docs/10/static/triggers.html
- Event Triggers (Chapter 39): https://www.postgresql.org/docs/10/static/event-triggers.html
- Trigger functions (Chapter 42.9): https://www.postgresql.org/docs/10/static/plpgsql-trigger.html
- CREATE TRIGGER: https://www.postgresql.org/docs/10/static/sql-createtrigger.html
- ALTER TRIGGER: https://www.postgresql.org/docs/10/static/sql-altertrigger.html
- DROP TRIGGER: https://www.postgresql.org/docs/10/static/sql-droptrigger.html

In addition there are plenty of articles in favour and against triggers on the internet. Listing them here is not meaningful, simply search for them with your preferred search engine.

Contact

- Work: clavadetscher@kof.ethz.ch
  http://www.kof.ethz.ch
- SwissPUG: clavadetscher@swisspug.org
  http://www.swisspug.org
- Private: charles@artesano.ch
  http://www.artesano.ch

Thank you

Thank you very much for your attention !

https://2018.pgday.paris/feedback

Q&A