Trigger and Rule Systems

- PostgreSQL provides triggers and rule systems to automatically perform a certain function when an event, such as `INSERT`, `UPDATE`, or `DELETE`, is performed.
- Triggers and rules cannot be defined on `SELECT` statements, except for `_RETURN`, which is used in the internal implementation of PostgreSQL views.

- From a functionality point of view, the trigger system is more generic; it can be used to implement complex actions more easily than rules.
- However, both trigger and rule systems can be used to implement the same functionalities in several cases.
- From a performance point of view, rules tend to be faster than triggers, but triggers tend to be simpler and more compatible with other RDBMs, since the rule system is a PostgreSQL extension.

Rule System

- Creating a rule will either rewire the default rule or create a new rule for specific action on a specific table or view.
- In other words, a rule on an `insert` action can change the `insert` action's behavior or create a new action for `insert`.
- When using the rule system, note that it is based on the C macro system.
- This means that you can get strange results when it is used with volatile functions, such as `random()`, and sequence functions, such as `nextval()`.

- The following example shows how tricky the rule system can be.
- Let's suppose that we would like to audit the table for `car`.
- For this reason, a new table, called `car_log`, will be created to keep a track of all of the actions on the `car` table, such as `UPDATE`, `DELETE`, and `INSERT`.

```
car_portal=# CREATE TABLE car_portal_app.car_log (LIKE car_portal_app.car);
CREATE TABLE
car_portal=# ALTER TABLE car_portal_app.car_log
car_portal-# ADD COLUMN car_log_action varchar (1) NOT NULL,
car_portal-# ADD COLUMN car_log_time TIMESTAMP WITH TIME ZONE NOT NULL;
ALTER TABLE
car_portal=# CREATE RULE car_log AS
car_portal-# ON INSERT TO car_portal_app.car DO ALSO
car_portal-# INSERT INTO car_portal_app.car_log (car_id, car_model_id, number_of_owners, registration_number, number_of_doors, manufacture_yea
r,car_log_action, car_log_time)
car_portal-# VALUES (new.car_id, new.car_model_id, new.number_of_owners, new.registration_number, new.number_of_doors, new.manufacture_year, '
I', now());
CREATE RULE
```

- To test the code, let's insert a record, as follows:

```
car_portal=> INSERT INTO car_portal_app.car (car_id, car_model_id, number_of_owners, registration_number, number_of_doors, manufacture_year)
car_portal-> VALUES (100000, 2, 2, 'x', 3, 2017);
INSERT 0 1
```

- You can examine the contents of the `car_log` table, as follows:

```
car_portal=> \x
Expanded display is on.
car_portal=> SELECT to_json(car_log)
car_portal-> FROM car_portal_app.car_log
car_portal-> WHERE registration_number ='x';
-[ RECORD 1 ]----------------------------------------------------------------------------------------------------------------------------------------------------------------
to_json | {"car_id":100000,"number_of_owners":2,"registration_number":"x","manufacture_year":2017,"number_of_doors":3,"car_model_id":2,"mileag
e":null,"car_log_action":"I","car_log_time":"2019-03-07T22:12:02.411222+08:00"}

car_portal=> SELECT to_json(car)
car_portal-> FROM car_portal_app.car
car_portal-> WHERE registration_number ='x';
-[ RECORD 1 ]----------------------------------------------------------------------------------------------------------------------------------------------------------------
to_json | {"car_id":100000,"number_of_owners":2,"registration_number":"x","manufacture_year":2017,"number_of_doors":3,"car_model_id":2,"mileag
e":null}
```

- However, as mentioned earlier, the rules system is built on macros, which cause some issues.
- For example, if we use the default value, it will create a problem, as shown in the following code snippet:

```
car_portal=> INSERT INTO car (car_id, car_model_id, number_of_owners, registration_number, number_of_doors, manufacture_year)
car_portal-> VALUES (default, 2, 2, 'y', 3, 2017);
INSERT 0 1
```

- Now, we can test the contents of the two tables, as follows:

```
car_portal=> SELECT to_json(car)
car_portal-> FROM car
car_portal-> WHERE registration_number ='y';
-[ RECORD 1 ]-------------------------------------------------------------------------------------------------------------------------------------------
-----
to_json | {"car_id":230,"number_of_owners":2,"registration_number":"y","manufacture_year":2017,"number_of_doors":3,"car_model_id":2,"mileage":
null}

car_portal=> SELECT to_json(car_log)
car_portal-> FROM car_log
car_portal-> WHERE registration_number ='y';
-[ RECORD 1 ]-------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------
to_json | {"car_id":231,"number_of_owners":2,"registration_number":"y","manufacture_year":2017,"number_of_doors":3,"car_model_id":2,"mileage":
null,"car_log_action":"I","car_log_time":"2019-03-07T22:30:30.936224+08:00"}
```

- Note that the records in the `car` table and the `car_log` table are not identical.
- `car_id` in the `car` table is less than `car_id` in the `car_log` table, by one.
- We can see that the `DEFAULT` keyword is used to assign `car_id` the default value, which is `nextval('car_car_id_seq'::regclass)`.

- When you are creating a rule, you can have a conditional rule (that is, you can rewrite the action if a certain condition is met), as shown in the following rule synopsis.
  - ```
    CREATE [ OR REPLACE ] RULE name AS ON event
        TO table_name [ WHERE condition ]
        DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
    ```
- However, you cannot have conditional rules for `INSERT`, `UPDATE`, and `DELETE` on views without having unconditional rules.
- To solve this problem, you can create an unconditional dummy rule.
- Having rules on views is one way of implementing updatable views.
    ```

Trigger System

- PostgreSQL triggers a function when a certain event occurs on a `table`, `view`, or `foreign` table.
- Triggers are executed when a user tries to modify the data through any of the DML events, including `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE`.

- The trigger synopsis is as follows:

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_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
```

- The trigger time context is one of the following:
  - `BEFORE`: This is only applied to tables and is fired before the constraints are checked and the operation is performed. It is useful for checking data constraints on several tables, when it is not possible to model using referential integrity constraints.
  - `AFTER`: This is also only applied on tables and is fired after the operation is performed. It is useful for cascading changes to other tables. An example use case would be data auditing.
  - `INSTEAD OF`: This is applied on views and is used to make views updatable.

- When a trigger is marked for each row, then the trigger function will be executed for each row that has been affected by the CRUD operation.
- A statement trigger is only executed once per operation.
- When the `WHEN` condition is supplied, only the rows that fulfill the condition will be handled by the trigger.

- Finally, triggers can be marked as `CONSTRAINT`, to control when they can be executed; a trigger can be executed after the end of the statement or at the end of the transaction.
- The constraint trigger must be the `AFTER` and `FOR EACH ROW` trigger, and the firing time of the constraint trigger is controlled by the following options:
  - `DEFERRABLE`: This marks the trigger as deferrable, which can be used to cause the trigger firing to be postponed till the end of the transaction.
  - `INITIALLY DEFERRED`: This specifies the time when the trigger is to be executed. This means that the trigger will be executed at the end of the transaction. The trigger should be marked as `DEFERRABLE`.
  - `NOT DEFERRABLE`: This is the default behavior of the trigger, which will cause the trigger to be fired after each statement in the transaction.
  - `INITIALLY IMMEDIATE`: This specifies the time when the trigger is to be executed. This means that the trigger will be executed after each statement. The trigger should be marked as `DEFERRABLE`.

- These options are very useful when PostgreSQL interacts with external systems, such as `memcached`.
- For example, let's suppose that we have a trigger on a table, and this table is cached; whenever the table is updated, the cache is also updated.
- Since the caching system is not transactional, we can postpone the update until the end of the transaction in order to guarantee data consistency.

- SET CONSTRAINTS
  - SET CONSTRAINTS { ALL | *name* [, ...] } { DEFERRED | IMMEDIATE }
  - SET CONSTRAINTS sets the behavior of constraint checking within the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are not checked until transaction commit. Each constraint has its own IMMEDIATE or DEFERRED mode.

- Trigger names define the execution order of the triggers, which have the same firing time context alphabetically.

- To explain the trigger system, let's redo the `car_log` table example using triggers.
- First of all, notice that the trigger type is the `AFTER` trigger, since the data should first be checked against the car table constraint before inserting it into the new table.
- To create a trigger, you need to create a function, as follows:

```
car_portal=> CREATE OR REPLACE FUNCTION car_portal_app.car_log_trg () RETURNS TRIGGER AS $$
car_portal$> BEGIN
car_portal$> IF TG_OP = 'INSERT' THEN
car_portal$>     INSERT INTO car_portal_app.car_log SELECT NEW.*, 'I', NOW();
car_portal$> ELSIF TG_OP = 'UPDATE' THEN
car_portal$>     INSERT INTO car_portal_app.car_log SELECT NEW.*, 'U', NOW();
car_portal$> ELSIF TG_OP = 'DELETE' THEN
car_portal$>     INSERT INTO car_portal_app.car_log SELECT OLD.*, 'D', NOW();
car_portal$> END IF;
car_portal$> RETURN NULL; --ignored since this is after trigger
car_portal$> END;
car_portal$> $$
car_portal-> LANGUAGE plpgsql;
CREATE FUNCTION
```

- To create `TRIGGER`, we need to execute the following statement:

```
car_portal=> CREATE TRIGGER car_log
car_portal-> AFTER INSERT OR UPDATE OR DELETE ON car_portal_app.car
car_portal-> FOR EACH ROW EXECUTE PROCEDURE car_portal_app.car_log_trg ();
CREATE TRIGGER
```

```
car_portal=> INSERT INTO car (car_id, car_model_id, number_of_owners, registration_number, number_of_doors, manufacture_year)
car_portal-> VALUES (default, 2, 2, 'z', 3, 2019);
INSERT 0 1
car_portal=> SELECT to_json(car)
car_portal-> FROM car
car_portal-> WHERE registration_number ='z';
-[ RECORD 1 ]-----------------------------------------------------------------------------------------------------------------------------------------
-----
to_json | {"car_id":232,"number_of_owners":2,"registration_number":"z","manufacture_year":2019,"number_of_doors":3,"car_model_id":2,"mileage":
null}

car_portal=> SELECT to_json(car_log)
car_portal-> FROM car_log
car_portal-> WHERE registration_number ='z';
-[ RECORD 1 ]-----------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------
to_json | {"car_id":232,"number_of_owners":2,"registration_number":"z","manufacture_year":2019,"number_of_doors":3,"car_model_id":2,"mileage":
null,"car_log_action":"I","car_log_time":"2019-03-08T15:13:29.04813+08:00"}
```

- The trigger function should fulfill the following requirements:
  - **Return type**: The TRIGGER function should return the TRIGGER pseudotype.
  - **Return value**: The TRIGGER function must return a value.
    Row-level triggers fired `BEFORE` can return `NULL` to signal the trigger manager to skip the rest of the operation for this row (i.e., subsequent triggers are not fired, and the `INSERT`/`UPDATE`/`DELETE` does not occur for this row). If a nonnull value is returned then the operation proceeds with that row value.
    The return value of a row-level trigger fired `AFTER` or a statement-level trigger fired `BEFORE` or `AFTER` is always ignored.
  - **No arguments**: The TRIGGER function must be declared without an argument, even if you need to pass an argument to it.

- The passing of an argument is achieved via the `TG_ARG` variable.
- When the trigger function is created, several variables, such as `TG_ARG` and `NEW`, are created automatically.
- Other variables that are created are listed in the following table:

Trigger variable	Data type	Description
NEW	RECORD	This holds the row that is inserted or updated. In the case of the statement-level trigger, it is NULL.
OLD	RECORD	This holds the old row that is updated or deleted. In the case of the statement-level trigger, it is NULL.

TG_NAME	NAME	This is the trigger name.
TG_OP	NAME	This is the trigger operation, which can have one of the following values—INSERT, UPDATE, DELETE, or TRUNCATE.
TG_WHEN	NAME	This is the time when the trigger is fired, which can have one of the following values—AFTER or BEFORE.
TG_RELID	OID	This is the relation OID. You can get the relation name by casting it to text, using regclass::text.

TG_TABLE_NAME	NAME	This is the trigger table name.
TG_TABLE_SCHEMA	NAME	This is the trigger table schema name.
TG_ARG[]	TEXT array	This is the trigger argument. The indexing starts from zero, and a wrong index returns NULL.
TG_NARG	INTEGER	This is the number of arguments passed to the trigger.

- A row-level trigger, which is fired `BEFORE` the actual operation, returning null values, will cancel the operation.
  - This means that the next trigger will not be fired, and the affected row will not be deleted, updated, or inserted.
- For a trigger that's fired `AFTER` the operation or for a statement-level trigger, the return value will be ignored; however, the operation will be aborted if the trigger function raises an exception or an error, due to the relational database's transactional behavior.

- In the preceding auditing example, if we change the `car` table definition (for example, by adding or dropping a column), the trigger function on the `car` table will fail, leading to the ignoring of the newly inserted or updated row.
- You can solve this by using exception trapping in the trigger definition.

Triggers with Arguments

- In the following example, another general auditing technique will be presented, which can be applied to several tables, while some table columns can be excluded from auditing.
- The new editing techniques use the `hstore` extension.
- `hstore` defines a hash map data type and provides a set of functions and operators to handle this data type.
- In the new auditing technique, the table rows will be stored as a hash map.

- The first step is to create the `hstore` extension and a table where the audited data will be stored, as follows:

```
car_portal=> reset role;
RESET
car_portal=# select current_user;
-[ RECORD 1 ]+---------
current_user | postgres

car_portal=# CREATE extension hstore;
CREATE EXTENSION
car_portal=# set role car_portal_app;
SET
car_portal=> SET search_path to car_portal_app;
SET
car_portal=> CREATE TABLE car_portal_app.log(
car_portal(>     schema_name text NOT NULL,
car_portal(>     table_name text NOT NULL,
car_portal(>     old_row hstore,
car_portal(>     new_row hstore,
car_portal(>     action TEXT check (action IN ('I','U','D')) NOT NULL,
car_portal(>     created_by text NOT NULL,
car_portal(>     created_on timestamp without time zone NOT NULL
car_portal(> );
CREATE TABLE
```

- The second step is to define the `TRIGGER` function, as follows:

```
car_portal=> CREATE OR REPLACE FUNCTION car_portal_app.log_audit() RETURNS trigger AS $$
car_portal$> DECLARE
car_portal$>     log_row car_portal_app.log;
car_portal$>     excluded_columns text[] = NULL;
car_portal$> BEGIN
car_portal$>     log_row = ROW (TG_TABLE_SCHEMA::text, TG_TABLE_NAME::text,NULL,NULL,NULL,current_user::TEXT,current_timestamp);
car_portal$>
car_portal$>     IF TG_ARGV[0] IS NOT NULL THEN
car_portal$>         excluded_columns = TG_ARGV[0]::text[];
car_portal$>     END IF;
car_portal$>     IF (TG_OP = 'INSERT') THEN
car_portal$>         log_row.new_row = hstore(NEW.*) - excluded_columns;
car_portal$>         log_row.action ='I';
car_portal$>     ELSIF (TG_OP = 'UPDATE' AND (hstore(OLD.*) - excluded_columns != hstore(NEW.*) - excluded_columns)) THEN
car_portal$>         log_row.old_row = hstore(OLD.*) - excluded_columns;
car_portal$>         log_row.new_row = hstore(NEW.* )- excluded_columns;
car_portal$>         log_row.action ='U';
car_portal$>     ELSIF (TG_OP = 'DELETE') THEN
car_portal$>         log_row.old_row = hstore (OLD.*) - excluded_columns;
car_portal$>         log_row.action ='D';
car_portal$>     ELSE
car_portal$>         RETURN NULL; -- update on excluded columns
car_portal$>     END IF;
car_portal$>     INSERT INTO car_portal_app.log SELECT log_row.*;
car_portal$>     RETURN NULL;
car_portal$> END;
car_portal$> $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

- To apply the preceding trigger to the `car` table, assuming that the `number_of_doors` attribute should be excluded from tracking, we can create the trigger as follows:

```
car_portal=> CREATE TRIGGER car_log_trg
car_portal-> AFTER INSERT OR UPDATE OR DELETE ON car_portal_app.car
car_portal-> FOR EACH ROW EXECUTE PROCEDURE log_audit('{number_of_doors}');
CREATE TRIGGER
```

- The following example shows the trigger behavior for the insert statement:

```
car_portal=> INSERT INTO car (car_id, car_model_id, number_of_owners, registration_number, number_of_doors, manufacture_year)
car_portal-> VALUES (default, 2, 2, 'zz', 3, 2017);
INSERT 0 1
```

- To display the results, let's retrieve the contents of the `log` table in a JSON format, as follows:

```
car_portal=> SELECT jsonb_pretty((to_json(log))::jsonb)
car_portal-> FROM car_portal_app.log
car_portal-> WHERE action = 'I' and new_row->'registration_number'='zz';
-[ RECORD 1 ]+-----------------------------------------------
jsonb_pretty | {                                              +
             |     "action": "I",                             +
             |     "new_row": {                               +
             |         "car_id": "234",                       +
             |         "mileage": null,                       +
             |         "car_model_id": "2",                   +
             |         "manufacture_year": "2017",            +
             |         "number_of_owners": "2",               +
             |         "registration_number": "zz"            +
             |     },                                         +
             |     "old_row": null,                           +
             |     "created_by": "car_portal_app",            +
             |     "created_on": "2019-03-08T18:25:22.965655",+
             |     "table_name": "car",                       +
             |     "schema_name": "car_portal_app"            +
             | }
```

Triggers and Updatable Views

- For views that are not automatically updatable, the trigger system can be used to make them updatable.
- The `seller_account_information` view, which shows the information about the seller account, is not automatically updatable because it has `INNER JOIN`:

```
car_portal=> CREATE OR REPLACE VIEW car_portal_app.seller_account_info AS
car_portal->   SELECT account.account_id, first_name, last_name, email, password, seller_account_id,
car_portal->     total_rank, number_of_advertisement, street_name, street_number, zip_code , city
car_portal->   FROM car_portal_app.account INNER JOIN
car_portal->     car_portal_app.seller_account ON (account.account_id = seller_account.account_id);
CREATE VIEW
```

- To verify that the preceding view, `seller_account_information`, is not updatable, we can check the information schema, as follows:

```
car_portal=> SELECT is_insertable_into
car_portal-> FROM information_schema.tables
car_portal-> WHERE table_name = 'seller_account_info';
 is_insertable_into
--------------------
 NO
(1 row)
```

- The following trigger function assumes that `account_id` and `seller_account_id` are always generated using the default values, which are the sequences generated automatically when creating a serial data type.

```
car_portal=> CREATE OR REPLACE FUNCTION car_portal_app.seller_account_info_update () RETURNS TRIGGER AS $$
car_portal$> DECLARE
car_portal$>     acc_id INT;
car_portal$>     seller_acc_id INT;
car_portal$> BEGIN
car_portal$>     IF (TG_OP = 'INSERT') THEN
car_portal$>         WITH
car_portal$>         inserted_account AS (
car_portal$>             INSERT INTO car_portal_app.account (account_id, first_name, last_name, password, email)
car_portal$>             VALUES (DEFAULT, NEW.first_name, NEW.last_name, NEW.password, NEW.email)
car_portal$>             RETURNING account_id),
car_portal$>         inserted_seller_account AS (
car_portal$>             INSERT INTO car_portal_app.seller_account(seller_account_id, account_id, total_rank, number_of_advertisement,
car_portal$>                                                       street_name, street_number, zip_code, city)
car_portal$>             SELECT nextval('car_portal_app.seller_account_seller_account_id_seq'::regclass), account_id, NEW.total_rank,
car_portal$>                    NEW.number_of_advertisement, NEW.street_name, NEW.street_number, NEW.zip_code, NEW.city
car_portal$>             FROM inserted_account
car_portal$>             RETURNING account_id, seller_account_id)
car_portal$>         SELECT account_id, seller_account_id
car_portal$>         INTO acc_id, seller_acc_id
car_portal$>         FROM inserted_seller_account;
car_portal$>         NEW.account_id = acc_id;
car_portal$>         NEW.seller_account_id = seller_acc_id;
car_portal$>         RETURN NEW;
car_portal$>     ELSIF (TG_OP = 'UPDATE' AND OLD.account_id = NEW.account_id AND OLD.seller_account_id = NEW.seller_account_id) THEN
car_portal$>         UPDATE car_portal_app.account
car_portal$>         SET first_name = new.first_name, last_name = new.last_name, password= new.password, email = new.email
car_portal$>         WHERE account_id = new.account_id;
car_portal$>         UPDATE car_portal_app.seller_account
car_portal$>         SET total_rank = NEW.total_rank, number_of_advertisement = NEW.number_of_advertisement,
car_portal$>             street_name = NEW.street_name, street_number = NEW.street_number, zip_code = NEW.zip_code,
car_portal$>             city = NEW.city
car_portal$>         WHERE seller_account_id = NEW.seller_account_id;
car_portal$>         RETURN NEW;
car_portal$>     ELSIF (TG_OP = 'DELETE') THEN
car_portal$>         DELETE FROM car_portal_app.seller_account
car_portal$>         WHERE seller_account_id = OLD.seller_account_id;
car_portal$>         DELETE FROM car_portal_app.account
car_portal$>         WHERE account_id = OLD.account_id;
car_portal$>         RETURN OLD;
car_portal$>     ELSE
car_portal$>         RAISE EXCEPTION 'An error occurred for % operation', TG_OP;
car_portal$>         RETURN NULL;
car_portal$>     END IF;
car_portal$> END;
car_portal$> $$ LANGUAGE plpgsql;
CREATE FUNCTION
```

- To make the view updatable, you need to create an `INSTEAD OF` trigger and define the actions, as follows:

```
car_portal=> CREATE TRIGGER seller_account_info_trg
car_portal-> INSTEAD OF INSERT OR UPDATE OR DELETE ON car_portal_app.seller_account_info
car_portal-> FOR EACH ROW EXECUTE PROCEDURE car_portal_app.seller_account_info_update ();
CREATE TRIGGER
```

- To test `INSERT` on the view, we can run the following code:

```
car_portal=> INSERT INTO car_portal_app.seller_account_info (first_name,last_name, password, email, total_rank, number_of_advertisement,
car_portal(>                                                 street_name, street_number, zip_code, city)
car_portal-> VALUES ('test_first_name', 'test_last_name', 'test_password', 'test_email@test.com', NULL, 0, 'test_street_name',
car_portal(>         'test_street_number', 'test_zip_code','test_city')
car_portal-> RETURNING account_id, seller_account_id;
 account_id | seller_account_id
------------+-------------------
        484 |               148
(1 row)
```

- To test `DELETE` and `UPDATE`, we simply run the following snippet:

```
car_portal=> UPDATE car_portal_app.seller_account_info
car_portal-> SET email = 'test2@test.com'
car_portal-> WHERE seller_account_id=147
car_portal-> RETURNING seller_account_id;
 seller_account_id
-------------------
               147
(1 row)

UPDATE 1
car_portal=> DELETE FROM car_portal_app.seller_account_info
car_portal-> WHERE seller_account_id=147;
DELETE 1
```

- Finally, if we try to delete all seller accounts, it will fail due to a referential integrity constraint, as follows:

```
car_portal=> DELETE FROM car_portal_app.seller_account_info;
ERROR:  update or delete on table "seller_account" violates foreign key constraint "advertisement_seller_account_id_fkey" on table "advertisem
ent"
DETAIL:  Key (seller_account_id)=(57) is still referenced from table "advertisement".
CONTEXT:  SQL statement "DELETE FROM car_portal_app.seller_account
        WHERE seller_account_id = OLD.seller_account_id"
PL/pgSQL function seller_account_info_update() line 36 at SQL statement
```