NAME

CREATE_TRIGGER − define a new trigger

SYNOPSIS

```
CREATE [ OR REPLACE ] [ 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 { FUNCTION | PROCEDURE } function_name ( arguments )
```

where *event* can be one of:

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

DESCRIPTION

CREATE TRIGGER creates a new trigger. **CREATE OR REPLACE TRIGGER** will either create a new trigger, or replace an existing trigger. The trigger will be associated with the specified table, view, or foreign table and will execute the specified function *function_name* when certain operations are performed on that table.

To replace the current definition of an existing trigger, use **CREATE OR REPLACE TRIGGER**, specifying the existing trigger's name and parent table. All other properties are replaced.

The trigger can be specified to fire before the operation is attempted on a row (before constraints are checked and the **INSERT**, **UPDATE**, or **DELETE** is attempted); or after the operation has completed (after constraints are checked and the **INSERT**, **UPDATE**, or **DELETE** has completed); or instead of the operation (in the case of inserts, updates or deletes on a view). If the trigger fires before or instead of the event, the trigger can skip the operation for the current row, or change the row being inserted (for **INSERT** and **UPDATE** operations only). If the trigger fires after the event, all changes, including the effects of other triggers, are “visible” to the trigger.

A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. For example, a **DELETE** that affects 10 rows will cause any ON DELETE triggers on the target relation to be called 10 separate times, once for each deleted row. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies (in particular, an operation that modifies zero rows will still result in the execution of any applicable FOR EACH STATEMENT triggers).

Triggers that are specified to fire INSTEAD OF the trigger event must be marked FOR EACH ROW, and can only be defined on views. BEFORE and AFTER triggers on a view must be marked as FOR EACH STATEMENT.

In addition, triggers may be defined to fire for **TRUNCATE**, though only FOR EACH STATEMENT.

The following table summarizes which types of triggers may be used on tables, views, and foreign tables:

When	Event	Row−level	Statement−level
BEFORE	**INSERT/UPDATE/DELETE**	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables and foreign tables
AFTER	**INSERT/UPDATE/DELETE**	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables and foreign tables
INSTEAD OF	**INSERT/UPDATE/DELETE**	Views	—
	TRUNCATE	—	—

Also, a trigger definition can specify a Boolean WHEN condition, which will be tested to see whether the trigger should be fired. In row−level triggers the WHEN condition can examine the old and/or new values of columns of the row. Statement−level triggers can also have WHEN conditions, although the feature is not so useful for them since the condition cannot refer to any values in the table.

If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.

When the CONSTRAINT option is specified, this command creates a constraint trigger. This is the same as a regular trigger except that the timing of the trigger firing can be adjusted using **SET CONSTRAINTS**. Constraint triggers must be AFTER ROW triggers on plain tables (not foreign tables). They can be fired either at the end of the statement causing the triggering event, or at the end of the containing transaction; in the latter case they are said to be deferred. A pending deferred−trigger firing can also be forced to happen immediately by using **SET CONSTRAINTS**. Constraint triggers are expected to raise an exception when the constraints they implement are violated.

The REFERENCING option enables collection of transition relations, which are row sets that include all of the rows inserted, deleted, or modified by the current SQL statement. This feature lets the trigger see a global view of what the statement did, not just one row at a time. This option is only allowed for an AFTER trigger on a plain table (not a foreign table). The trigger should not be a constraint trigger. Also, if the trigger is an UPDATE trigger, it must not specify a *column_name* list when using this option. OLD TABLE may only be specified once, and only for a trigger that can fire on UPDATE or DELETE; it creates a transition relation containing the before−images of all rows updated or deleted by the statement. Similarly, NEW TABLE may only be specified once, and only for a trigger that can fire on UPDATE or INSERT; it creates a transition relation containing the after−images of all rows updated or inserted by the statement.

SELECT does not modify any rows so you cannot create **SELECT** triggers. Rules and views may provide workable solutions to problems that seem to need **SELECT** triggers.

Refer to Chapter 37 for more information about triggers.

PARAMETERS

name

The name to give the new trigger. This must be distinct from the name of any other trigger for the same table. The name cannot be schema−qualified — the trigger inherits the schema of its table. For a constraint trigger, this is also the name to use when modifying the trigger's behavior using **SET CONSTRAINTS**.

BEFORE
AFTER
INSTEAD OF

Determines whether the function is called before, after, or instead of the event. A constraint trigger can only be specified as AFTER.

event

One of INSERT, UPDATE, DELETE, or TRUNCATE; this specifies the event that will fire the trigger.

Multiple events can be specified using OR, except when transition relations are requested.

For UPDATE events, it is possible to specify a list of columns using this syntax:

UPDATE OF *column_name1* [, *column_name2* ...]

The trigger will only fire if at least one of the listed columns is mentioned as a target of the **UPDATE** command or if one of the listed columns is a generated column that depends on a column that is the target of the **UPDATE**.

INSTEAD OF UPDATE events do not allow a list of columns. A column list cannot be specified when requesting transition relations, either.

table_name
The name (optionally schema−qualified) of the table, view, or foreign table the trigger is for.

referenced_table_name
The (possibly schema−qualified) name of another table referenced by the constraint. This option is used for foreign−key constraints and is not recommended for general use. This can only be specified for constraint triggers.

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED
The default timing of the trigger. See the CREATE TABLE (**CREATE_TABLE**(7)) documentation for details of these constraint options. This can only be specified for constraint triggers.

REFERENCING
This keyword immediately precedes the declaration of one or two relation names that provide access to the transition relations of the triggering statement.

OLD TABLE
NEW TABLE
This clause indicates whether the following relation name is for the before−image transition relation or the after−image transition relation.

transition_relation_name
The (unqualified) name to be used within the trigger for this transition relation.

FOR EACH ROW
FOR EACH STATEMENT
This specifies whether the trigger function should be fired once for every row affected by the trigger event, or just once per SQL statement. If neither is specified, FOR EACH STATEMENT is the default. Constraint triggers can only be specified FOR EACH ROW.

condition
A Boolean expression that determines whether the trigger function will actually be executed. If WHEN is specified, the function will only be called if the *condition* returns true. In FOR EACH ROW triggers, the WHEN condition can refer to columns of the old and/or new row values by writing OLD.*column_name* or NEW.*column_name* respectively. Of course, INSERT triggers cannot refer to OLD and DELETE triggers cannot refer to NEW.

INSTEAD OF triggers do not support WHEN conditions.

Currently, WHEN expressions cannot contain subqueries.

Note that for constraint triggers, evaluation of the WHEN condition is not deferred, but occurs immediately after the row update operation is performed. If the condition does not evaluate to true

then the trigger is not queued for deferred execution.

function_name
A user−supplied function that is declared as taking no arguments and returning type trigger, which is executed when the trigger fires.

In the syntax of CREATE TRIGGER, the keywords FUNCTION and PROCEDURE are equivalent, but the referenced function must in any case be a function, not a procedure. The use of the keyword PROCEDURE here is historical and deprecated.

arguments
An optional comma−separated list of arguments to be provided to the function when the trigger is executed. The arguments are literal string constants. Simple names and numeric constants can be written here, too, but they will all be converted to strings. Please check the description of the implementation language of the trigger function to find out how these arguments can be accessed within the function; it might be different from normal function arguments.

NOTES

To create or replace a trigger on a table, the user must have the TRIGGER privilege on the table. The user must also have EXECUTE privilege on the trigger function.

Use **DROP TRIGGER** to remove a trigger.

Creating a row−level trigger on a partitioned table will cause an identical “clone” trigger to be created on each of its existing partitions; and any partitions created or attached later will have an identical trigger, too. If there is a conflictingly−named trigger on a child partition already, an error occurs unless **CREATE OR REPLACE TRIGGER** is used, in which case that trigger is replaced with a clone trigger. When a partition is detached from its parent, its clone triggers are removed.

A column−specific trigger (one defined using the UPDATE OF *column_name* syntax) will fire when any of its columns are listed as targets in the **UPDATE** command's SET list. It is possible for a column's value to change even when the trigger is not fired, because changes made to the row's contents by BEFORE UPDATE triggers are not considered. Conversely, a command such as UPDATE ... SET x = x ... will fire a trigger on column x, even though the column's value did not change.

In a BEFORE trigger, the WHEN condition is evaluated just before the function is or would be executed, so using WHEN is not materially different from testing the same condition at the beginning of the trigger function. Note in particular that the NEW row seen by the condition is the current value, as possibly modified by earlier triggers. Also, a BEFORE trigger's WHEN condition is not allowed to examine the system columns of the NEW row (such as ctid), because those won't have been set yet.

In an AFTER trigger, the WHEN condition is evaluated just after the row update occurs, and it determines whether an event is queued to fire the trigger at the end of statement. So when an AFTER trigger's WHEN condition does not return true, it is not necessary to queue an event nor to re−fetch the row at end of statement. This can result in significant speedups in statements that modify many rows, if the trigger only needs to be fired for a few of the rows.

In some cases it is possible for a single SQL command to fire more than one kind of trigger. For instance an **INSERT** with an ON CONFLICT DO UPDATE clause may cause both insert and update operations, so it will fire both kinds of triggers as needed. The transition relations supplied to triggers are specific to their event type; thus an **INSERT** trigger will see only the inserted rows, while an **UPDATE** trigger will see only the updated rows.

Row updates or deletions caused by foreign−key enforcement actions, such as ON UPDATE CASCADE or ON DELETE SET NULL, are treated as part of the SQL command that caused them (note that such actions are never deferred). Relevant triggers on the affected table will be fired, so that this provides another way in which an SQL command might fire triggers not directly matching its type. In simple cases, triggers that request transition relations will see all changes caused in their table by a single original SQL command as a single transition relation. However, there are cases in which the presence of an AFTER ROW trigger that requests transition relations will cause the foreign−key enforcement actions triggered by a single SQL

command to be split into multiple steps, each with its own transition relation(s). In such cases, any statement−level triggers that are present will be fired once per creation of a transition relation set, ensuring that the triggers see each affected row in a transition relation once and only once.

Statement−level triggers on a view are fired only if the action on the view is handled by a row−level INSTEAD OF trigger. If the action is handled by an INSTEAD rule, then whatever statements are emitted by the rule are executed in place of the original statement naming the view, so that the triggers that will be fired are those on tables named in the replacement statements. Similarly, if the view is automatically updatable, then the action is handled by automatically rewriting the statement into an action on the view's base table, so that the base table's statement−level triggers are the ones that are fired.

Modifying a partitioned table or a table with inheritance children fires statement−level triggers attached to the explicitly named table, but not statement−level triggers for its partitions or child tables. In contrast, row−level triggers are fired on the rows in affected partitions or child tables, even if they are not explicitly named in the query. If a statement−level trigger has been defined with transition relations named by a REFERENCING clause, then before and after images of rows are visible from all affected partitions or child tables. In the case of inheritance children, the row images include only columns that are present in the table that the trigger is attached to.

Currently, row−level triggers with transition relations cannot be defined on partitions or inheritance child tables. Also, triggers on partitioned tables may not be INSTEAD OF.

Currently, the OR REPLACE option is not supported for constraint triggers.

Replacing an existing trigger within a transaction that has already performed updating actions on the trigger's table is not recommended. Trigger firing decisions, or portions of firing decisions, that have already been made will not be reconsidered, so the effects could be surprising.

There are a few built−in trigger functions that can be used to solve common problems without having to write your own trigger code; see Section 9.29.

EXAMPLES

Execute the function **check_account_update** whenever a row of the table accounts is about to be updated:

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

Modify that trigger definition to only execute the function if column balance is specified as a target in the **UPDATE** command:

```
CREATE OR REPLACE TRIGGER check_update
  BEFORE UPDATE OF balance ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

This form only executes the function if column balance has in fact changed value:

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
  EXECUTE FUNCTION check_account_update();
```

Call a function to log updates of accounts, but only if something changed:

```
CREATE TRIGGER log_update
  AFTER UPDATE ON accounts
```

```
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE FUNCTION log_account_update();
```

Execute the function **view_insert_row** for each row to insert rows into the tables underlying a view:

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE FUNCTION view_insert_row();
```

Execute the function **check_transfer_balances_to_zero** for each statement to confirm that the transfer rows offset to a net of zero:

```
CREATE TRIGGER transfer_insert
    AFTER INSERT ON transfer
    REFERENCING NEW TABLE AS inserted
    FOR EACH STATEMENT
    EXECUTE FUNCTION check_transfer_balances_to_zero();
```

Execute the function **check_matching_pairs** for each row to confirm that changes are made to matching pairs at the same time (by the same statement):

```
CREATE TRIGGER paired_items_update
    AFTER UPDATE ON paired_items
    REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
    FOR EACH ROW
    EXECUTE FUNCTION check_matching_pairs();
```

Section 37.4 contains a complete example of a trigger function written in C.

COMPATIBILITY

The **CREATE TRIGGER** statement in PostgreSQL implements a subset of the SQL standard. The following functionalities are currently missing:

- While transition table names for AFTER triggers are specified using the REFERENCING clause in the standard way, the row variables used in FOR EACH ROW triggers may not be specified in a REFERENCING clause. They are available in a manner that is dependent on the language in which the trigger function is written, but is fixed for any one language. Some languages effectively behave as though there is a REFERENCING clause containing OLD ROW AS OLD NEW ROW AS NEW.
- The standard allows transition tables to be used with column−specific UPDATE triggers, but then the set of rows that should be visible in the transition tables depends on the trigger's column list. This is not currently implemented by PostgreSQL.
- PostgreSQL only allows the execution of a user−defined function for the triggered action. The standard allows the execution of a number of other SQL commands, such as **CREATE TABLE**, as the triggered action. This limitation is not hard to work around by creating a user−defined function that executes the desired commands.

SQL specifies that multiple triggers should be fired in time−of−creation order. PostgreSQL uses name order, which was judged to be more convenient.

SQL specifies that BEFORE DELETE triggers on cascaded deletes fire *after* the cascaded DELETE completes. The PostgreSQL behavior is for BEFORE DELETE to always fire before the delete action, even a cascading one. This is considered more consistent. There is also nonstandard behavior if BEFORE triggers modify rows or prevent updates during an update that is caused by a referential action. This can lead to constraint violations or stored data that does not honor the referential constraint.

The ability to specify multiple actions for a single trigger using OR is a PostgreSQL extension of the SQL standard.

The ability to fire triggers for **TRUNCATE** is a PostgreSQL extension of the SQL standard, as is the ability to define statement−level triggers on views.

CREATE CONSTRAINT TRIGGER is a PostgreSQL extension of the SQL standard. So is the OR REPLACE option.

SEE ALSO

ALTER TRIGGER (**ALTER_TRIGGER**(7)), DROP TRIGGER (**DROP_TRIGGER**(7)), CREATE FUNCTION (**CREATE_FUNCTION**(7)), SET CONSTRAINTS (**SET_CONSTRAINTS**(7))