

Chapter 36. Triggers

This chapter provides general information about writing trigger functions. Trigger functions can be written in most of the available procedural languages, including PL/pgSQL (Chapter 39), PL/Tcl (Chapter 40), PL/Perl (Chapter 41), and PL/Python (Chapter 42). After reading this chapter, you should consult the chapter for your favorite procedural language to find out the language-specific details of writing a trigger in it.

It is also possible to write a trigger function in C, although most people find it easier to use one of the procedural languages. It is not currently possible to write a trigger function in the plain SQL function language.

36.1. Overview of Trigger Behavior

A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed. Triggers can be attached to both tables and views.

On tables, triggers can be defined to execute either before or after any `INSERT`, `UPDATE`, or `DELETE` operation, either once per modified row, or once per SQL statement. `UPDATE` triggers can moreover be set to fire only if certain columns are mentioned in the `SET` clause of the `UPDATE` statement. Triggers can also fire for `TRUNCATE` statements. If a trigger event occurs, the trigger's function is called at the appropriate time to handle the event.

On views, triggers can be defined to execute instead of `INSERT`, `UPDATE`, or `DELETE` operations. `INSTEAD OF` triggers are fired once for each row that needs to be modified in the view. It is the responsibility of the trigger's function to perform the necessary modifications to the underlying base tables and, where appropriate, return the modified row as it will appear in the view. Triggers on views can also be defined to execute once per SQL statement, before or after `INSERT`, `UPDATE`, or `DELETE` operations.

The trigger function must be defined before the trigger itself can be created. The trigger function must be declared as a function taking no arguments and returning type `trigger`. (The trigger function receives its input through a specially-passed `TriggerData` structure, not in the form of ordinary function arguments.)

Once a suitable trigger function has been created, the trigger is established with `CREATE TRIGGER`. The same trigger function can be used for multiple triggers.

PostgreSQL offers both *per-row* triggers and *per-statement* triggers. With a per-row trigger, the trigger function is invoked once for each row that is affected by the statement that fired the trigger. In contrast, a per-statement trigger is invoked only once when an appropriate statement is executed, regardless of the number of rows affected by that statement. In particular, a statement that affects zero rows will still result in the execution of any applicable per-statement triggers. These two types of triggers are sometimes called *row-level* triggers and *statement-level* triggers, respectively. Triggers on `TRUNCATE` may only be defined at statement level. On views, triggers that fire before or after may only be defined at statement level, while triggers that fire instead of an `INSERT`, `UPDATE`, or `DELETE` may only be defined at row level.

Triggers are also classified according to whether they fire *before*, *after*, or *instead of* the operation. These are referred to as `BEFORE` triggers, `AFTER` triggers, and `INSTEAD OF` triggers respectively. Statement-level `BEFORE` triggers naturally fire before the statement starts to do anything, while statement-level `AFTER` triggers fire at the very end of the statement. These types of triggers may be defined on tables or views. Row-level `BEFORE` triggers fire immediately before a particular row is operated on, while row-level `AFTER` triggers fire at the end of the statement (but before any statement-

level `AFTER` triggers). These types of triggers may only be defined on tables. Row-level `INSTEAD OF` triggers may only be defined on views, and fire immediately as each row in the view is identified as needing to be operated on.

Trigger functions invoked by per-statement triggers should always return `NULL`. Trigger functions invoked by per-row triggers can return a table row (a value of type `HeapTuple`) to the calling executor, if they choose. A row-level trigger fired before an operation has the following choices:

- It can return `NULL` to skip the operation for the current row. This instructs the executor to not perform the row-level operation that invoked the trigger (the insertion, modification, or deletion of a particular table row).
- For row-level `INSERT` and `UPDATE` triggers only, the returned row becomes the row that will be inserted or will replace the row being updated. This allows the trigger function to modify the row being inserted or updated.

A row-level `BEFORE` trigger that does not intend to cause either of these behaviors must be careful to return as its result the same row that was passed in (that is, the `NEW` row for `INSERT` and `UPDATE` triggers, the `OLD` row for `DELETE` triggers).

A row-level `INSTEAD OF` trigger should either return `NULL` to indicate that it did not modify any data from the view's underlying base tables, or it should return the view row that was passed in (the `NEW` row for `INSERT` and `UPDATE` operations, or the `OLD` row for `DELETE` operations). A nonnull return value is used to signal that the trigger performed the necessary data modifications in the view. This will cause the count of the number of rows affected by the command to be incremented. For `INSERT` and `UPDATE` operations, the trigger may modify the `NEW` row before returning it. This will change the data returned by `INSERT RETURNING` or `UPDATE RETURNING`, and is useful when the view will not show exactly the same data that was provided.

The return value is ignored for row-level triggers fired after an operation, and so they can return `NULL`.

If more than one trigger is defined for the same event on the same relation, the triggers will be fired in alphabetical order by trigger name. In the case of `BEFORE` and `INSTEAD OF` triggers, the possibly-modified row returned by each trigger becomes the input to the next trigger. If any `BEFORE` or `INSTEAD OF` trigger returns `NULL`, the operation is abandoned for that row and subsequent triggers are not fired (for that row).

A trigger definition can also 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.) 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. However, 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. `INSTEAD OF` triggers do not support `WHEN` conditions.

Typically, row-level `BEFORE` triggers are used for checking or modifying the data that will be inserted or updated. For example, a `BEFORE` trigger might be used to insert the current time into a `timestamp` column, or to check that two elements of the row are consistent. Row-level `AFTER` triggers are most sensibly used to propagate the updates to other tables, or make consistency checks against other tables. The reason for this division of labor is that an `AFTER` trigger can be certain it is seeing the final value of the row, while a `BEFORE` trigger cannot; there might be other `BEFORE` triggers firing after it. If you

have no specific reason to make a trigger `BEFORE` or `AFTER`, the `BEFORE` case is more efficient, since the information about the operation doesn't have to be saved until end of statement.

If a trigger function executes SQL commands then these commands might fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger; for example, an `INSERT` trigger might execute a command that inserts an additional row into the same table, causing the `INSERT` trigger to be fired again. It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios.

When a trigger is being defined, arguments can be specified for it. The purpose of including arguments in the trigger definition is to allow different triggers with similar requirements to call the same function. As an example, there could be a generalized trigger function that takes as its arguments two column names and puts the current user in one and the current time stamp in the other. Properly written, this trigger function would be independent of the specific table it is triggering on. So the same function could be used for `INSERT` events on any table with suitable columns, to automatically track creation of records in a transaction table for example. It could also be used to track last-update events if defined as an `UPDATE` trigger.

Each programming language that supports triggers has its own method for making the trigger input data available to the trigger function. This input data includes the type of trigger event (e.g., `INSERT` or `UPDATE`) as well as any arguments that were listed in `CREATE TRIGGER`. For a row-level trigger, the input data also includes the `NEW` row for `INSERT` and `UPDATE` triggers, and/or the `OLD` row for `UPDATE` and `DELETE` triggers. Statement-level triggers do not currently have any way to examine the individual row(s) modified by the statement.

36.2. Visibility of Data Changes

If you execute SQL commands in your trigger function, and these commands access the table that the trigger is for, then you need to be aware of the data visibility rules, because they determine whether these SQL commands will see the data change that the trigger is fired for. Briefly:

- Statement-level triggers follow simple visibility rules: none of the changes made by a statement are visible to statement-level triggers that are invoked before the statement, whereas all modifications are visible to statement-level `AFTER` triggers.
- The data change (insertion, update, or deletion) causing the trigger to fire is naturally *not* visible to SQL commands executed in a row-level `BEFORE` trigger, because it hasn't happened yet.
- However, SQL commands executed in a row-level `BEFORE` trigger *will* see the effects of data changes for rows previously processed in the same outer command. This requires caution, since the ordering of these change events is not in general predictable; a SQL command that affects multiple rows can visit the rows in any order.
- Similarly, a row-level `INSTEAD OF` trigger will see the effects of data changes made by previous firings of `INSTEAD OF` triggers in the same outer command.
- When a row-level `AFTER` trigger is fired, all data changes made by the outer command are already complete, and are visible to the invoked trigger function.

If your trigger function is written in any of the standard procedural languages, then the above statements apply only if the function is declared `VOLATILE`. Functions that are declared `STABLE` or `IMMUTABLE` will not see changes made by the calling command in any case.

Further information about data visibility rules can be found in Section 43.4. The example in Section 36.4 contains a demonstration of these rules.

36.3. Writing Trigger Functions in C

This section describes the low-level details of the interface to a trigger function. This information is only needed when writing trigger functions in C. If you are using a higher-level language then these details are handled for you. In most cases you should consider using a procedural language before writing your triggers in C. The documentation of each procedural language explains how to write a trigger in that language.

Trigger functions must use the “version 1” function manager interface.

When a function is called by the trigger manager, it is not passed any normal arguments, but it is passed a “context” pointer pointing to a `TriggerData` structure. C functions can check whether they were called from the trigger manager or not by executing the macro:

```
CALLED_AS_TRIGGER(fcinfo)
```

which expands to:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

If this returns true, then it is safe to cast `fcinfo->context` to type `TriggerData *` and make use of the pointed-to `TriggerData` structure. The function must *not* alter the `TriggerData` structure or any of the data it points to.

`struct TriggerData` is defined in `commands/trigger.h`:

```
typedef struct TriggerData
{
    NodeTag      type;
    TriggerEvent tg_event;
    Relation     tg_relation;
    HeapTuple    tg_trigtuple;
    HeapTuple    tg_newtuple;
    Trigger      *tg_trigger;
    Buffer        tg_trigtuplebuf;
    Buffer        tg_newtuplebuf;
} TriggerData;
```

where the members are defined as follows:

`type`

Always `T_TriggerData`.

`tg_event`

Describes the event for which the function is called. You can use the following macros to examine `tg_event`:

```
TRIGGER_FIRED_BEFORE(tg_event)
```

Returns true if the trigger fired before the operation.

`TRIGGER_FIRED_AFTER(tg_event)`

Returns true if the trigger fired after the operation.

`TRIGGER_FIRED_INSTEAD(tg_event)`

Returns true if the trigger fired instead of the operation.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Returns true if the trigger fired for a row-level event.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Returns true if the trigger fired for a statement-level event.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Returns true if the trigger was fired by an `INSERT` command.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Returns true if the trigger was fired by an `UPDATE` command.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Returns true if the trigger was fired by a `DELETE` command.

`TRIGGER_FIRED_BY_TRUNCATE(tg_event)`

Returns true if the trigger was fired by a `TRUNCATE` command.

`tg_relation`

A pointer to a structure describing the relation that the trigger fired for. Look at `utils/rel.h` for details about this structure. The most interesting things are `tg_relation->rd_att` (descriptor of the relation tuples) and `tg_relation->rd_rel->relname` (relation name; the type is not `char*` but `NameData`; use `SPI_getrelname(tg_relation)` to get a `char*` if you need a copy of the name).

`tg_trigtuple`

A pointer to the row for which the trigger was fired. This is the row being inserted, updated, or deleted. If this trigger was fired for an `INSERT` or `DELETE` then this is what you should return from the function if you don't want to replace the row with a different one (in the case of `INSERT`) or skip the operation.

`tg_newtuple`

A pointer to the new version of the row, if the trigger was fired for an `UPDATE`, and `NULL` if it is for an `INSERT` or a `DELETE`. This is what you have to return from the function if the event is an `UPDATE` and you don't want to replace this row by a different one or skip the operation.

`tg_trigger`

A pointer to a structure of type `Trigger`, defined in `utils/rel.h`:

```
typedef struct Trigger
{
    Oid          tgoid;
    char        *tgname;
    Oid          tgfoid;
    int16        tgtype;
    bool         tgenabled;
    bool         tgisinternal;
    Oid          tgconstrrelid;
```

```

Oid      tgconstrindid;
Oid      tgconstraint;
bool     tgdeferrable;
bool     tginitdeferred;
int16    tgnargs;
int16    tgnattr;
int16    *tgattr;
char     **tgargs;
char     *tgqual;
} Trigger;

```

where `tgname` is the trigger's name, `tgnargs` is the number of arguments in `tgargs`, and `tgargs` is an array of pointers to the arguments specified in the `CREATE TRIGGER` statement. The other members are for internal use only.

`tg_trigtuplebuf`

The buffer containing `tg_trigtuple`, or `InvalidBuffer` if there is no such tuple or it is not stored in a disk buffer.

`tg_newtuplebuf`

The buffer containing `tg_newtuple`, or `InvalidBuffer` if there is no such tuple or it is not stored in a disk buffer.

A trigger function must return either a `HeapTuple` pointer or a `NULL` pointer (*not* an SQL null value, that is, do not set `isNull` true). Be careful to return either `tg_trigtuple` or `tg_newtuple`, as appropriate, if you don't want to modify the row being operated on.

36.4. A Complete Trigger Example

Here is a very simple example of a trigger function written in C. (Examples of triggers written in procedural languages can be found in the documentation of the procedural languages.)

The function `trigf` reports the number of rows in the table `ttest` and skips the actual operation if the command attempts to insert a null value into the column `x`. (So the trigger acts as a not-null constraint but doesn't abort the transaction.)

First, the table definition:

```

CREATE TABLE ttest (
    x integer
);

```

This is the source code of the trigger function:

```

#include "postgres.h"
#include "executor/spi.h"      /* this is what you need to work with SPI */
#include "commands/trigger.h" /* ... and triggers */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

extern Datum trigf(PG_FUNCTION_ARGS);

```

```

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char        *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;

    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                     SPI_tuptable->tupdesc,
                                     1,
                                     &isnull));

    elog(INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)

```

```

    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}

```

After you have compiled the source code (see Section 35.9.6), declare the function and the triggers:

```

CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

```

Now you can test the operation of the trigger:

```

=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                ^^^^^^^
                                remember what we said about visibility.

INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                ^^^^^^
                                remember what we said about visibility.

INSERT 167794 1
=> SELECT * FROM ttest;
 x
---

```



```

1
2
(2 rows)

=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                ^^^^^^
                                remember what we said about visibility.

DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)

```

There are more complex examples in `src/test/regress/regress.c` and in `spi`.