NAME

CREATE_RULE − define a new rewrite rule

SYNOPSIS

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

where *event* can be one of:

```
    SELECT | INSERT | UPDATE | DELETE
```

DESCRIPTION

CREATE RULE defines a new rule applying to a specified table or view. **CREATE OR REPLACE RULE** will either create a new rule, or replace an existing rule of the same name for the same table.

The PostgreSQL rule system allows one to define an alternative action to be performed on insertions, updates, or deletions in database tables. Roughly speaking, a rule causes additional commands to be executed when a given command on a given table is executed. Alternatively, an INSTEAD rule can replace a given command by another, or cause a command not to be executed at all. Rules are used to implement SQL views as well. It is important to realize that a rule is really a command transformation mechanism, or command macro. The transformation happens before the execution of the command starts. If you actually want an operation that fires independently for each physical row, you probably want to use a trigger, not a rule. More information about the rules system is in Chapter 39.

Presently, ON SELECT rules can only be attached to views. Such a rule must be named "_RETURN", must be an unconditional INSTEAD rule, and must have an action that consists of a single **SELECT** command. This command defines the visible contents of the view. (The view itself is basically a dummy table with no storage.) It's best to regard such a rule as an implementation detail. While a view can be redefined via CREATE OR REPLACE RULE "_RETURN" AS ..., it's better style to use CREATE OR REPLACE VIEW.

You can create the illusion of an updatable view by defining ON INSERT, ON UPDATE, and ON DELETE rules (or any subset of those that's sufficient for your purposes) to replace update actions on the view with appropriate updates on other tables. If you want to support **INSERT RETURNING** and so on, then be sure to put a suitable RETURNING clause into each of these rules.

There is a catch if you try to use conditional rules for complex view updates: there *must* be an unconditional INSTEAD rule for each action you wish to allow on the view. If the rule is conditional, or is not INSTEAD, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table of the view in some cases. If you want to handle all the useful cases in conditional rules, add an unconditional DO INSTEAD NOTHING rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules non−INSTEAD; in the cases where they are applied, they add to the default INSTEAD NOTHING action. (This method does not currently work to support RETURNING queries, however.)

Note

A view that is simple enough to be automatically updatable (see CREATE VIEW (**CREATE_VIEW**(7))) does not require a user−created rule in order to be updatable. While you can create an explicit rule anyway, the automatic update transformation will generally outperform an explicit rule.

Another alternative worth considering is to use INSTEAD OF triggers (see CREATE TRIGGER (**CREATE_TRIGGER**(7))) in place of rules.

PARAMETERS

name

The name of a rule to create. This must be distinct from the name of any other rule for the same table. Multiple rules on the same table and same event type are applied in alphabetical name order.

event
The event is one of SELECT, INSERT, UPDATE, or DELETE. Note that an **INSERT** containing an ON CONFLICT clause cannot be used on tables that have either INSERT or UPDATE rules. Consider using an updatable view instead.

table_name
The name (optionally schema−qualified) of the table or view the rule applies to.

condition
Any SQL conditional expression (returning boolean). The condition expression cannot refer to any tables except NEW and OLD, and cannot contain aggregate functions.

INSTEAD
INSTEAD indicates that the commands should be executed *instead of* the original command.

ALSO
ALSO indicates that the commands should be executed *in addition to* the original command.

If neither ALSO nor INSTEAD is specified, ALSO is the default.

command
The command or commands that make up the rule action. Valid commands are **SELECT**, **INSERT**, **UPDATE**, **DELETE**, or **NOTIFY**.

Within *condition* and *command*, the special table names NEW and OLD can be used to refer to values in the referenced table. NEW is valid in ON INSERT and ON UPDATE rules to refer to the new row being inserted or updated. OLD is valid in ON UPDATE and ON DELETE rules to refer to the existing row being updated or deleted.

NOTES

You must be the owner of a table to create or change rules for it.

In a rule for INSERT, UPDATE, or DELETE on a view, you can add a RETURNING clause that emits the view's columns. This clause will be used to compute the outputs if the rule is triggered by an **INSERT RETURNING**, **UPDATE RETURNING**, or **DELETE RETURNING** command respectively. When the rule is triggered by a command without RETURNING, the rule's RETURNING clause will be ignored. The current implementation allows only unconditional INSTEAD rules to contain RETURNING; furthermore there can be at most one RETURNING clause among all the rules for the same event. (This ensures that there is only one candidate RETURNING clause to be used to compute the results.) RETURNING queries on the view will be rejected if there is no RETURNING clause in any available rule.

It is very important to take care to avoid circular rules. For example, though each of the following two rule definitions are accepted by PostgreSQL, the **SELECT** command would cause PostgreSQL to report an error because of recursive expansion of a rule:

```
CREATE RULE "_RETURN" AS
  ON SELECT TO t1
  DO INSTEAD
    SELECT * FROM t2;

CREATE RULE "_RETURN" AS
  ON SELECT TO t2
  DO INSTEAD
    SELECT * FROM t1;

SELECT * FROM t1;
```

Presently, if a rule action contains a **NOTIFY** command, the **NOTIFY** command will be executed unconditionally, that is, the **NOTIFY** will be issued even if there are not any rows that the rule should apply to. For example, in:

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

one **NOTIFY** event will be sent during the **UPDATE**, whether or not there are any rows that match the condition id = 42. This is an implementation restriction that might be fixed in future releases.

COMPATIBILITY

CREATE RULE is a PostgreSQL language extension, as is the entire query rewrite system.

SEE ALSO

ALTER RULE (**ALTER_RULE**(7)), DROP RULE (**DROP_RULE**(7))