NAME

CREATE_POLICY − define a new row−level security policy for a table

SYNOPSIS

CREATE POLICY *name* ON *table_name*
[AS { PERMISSIVE | RESTRICTIVE }]
[FOR { ALL | SELECT | INSERT | UPDATE | DELETE }]
[TO { *role_name* | PUBLIC | CURRENT_ROLE | CURRENT_USER | SESSION_USER } [, ...]]
[USING (*using_expression*)]
[WITH CHECK (*check_expression*)]

DESCRIPTION

The **CREATE POLICY** command defines a new row−level security policy for a table. Note that row−level security must be enabled on the table (using **ALTER TABLE ... ENABLE ROW LEVEL SECURITY**) in order for created policies to be applied.

A policy grants the permission to select, insert, update, or delete rows that match the relevant policy expression. Existing table rows are checked against the expression specified in USING, while new rows that would be created via INSERT or UPDATE are checked against the expression specified in WITH CHECK. When a USING expression returns true for a given row then that row is visible to the user, while if false or null is returned then the row is not visible. Typically, no error occurs when a row is not visible, but see Table 300 for exceptions. When a WITH CHECK expression returns true for a row then that row is inserted or updated, while if false or null is returned then an error occurs.

For **INSERT**, **UPDATE**, and **MERGE** statements, WITH CHECK expressions are enforced after BEFORE triggers are fired, and before any actual data modifications are made. Thus a BEFORE ROW trigger may modify the data to be inserted, affecting the result of the security policy check. WITH CHECK expressions are enforced before any other constraints.

Policy names are per−table. Therefore, one policy name can be used for many different tables and have a definition for each table which is appropriate to that table.

Policies can be applied for specific commands or for specific roles. The default for newly created policies is that they apply for all commands and roles, unless otherwise specified. Multiple policies may apply to a single command; see below for more details. Table 300 summarizes how the different types of policy apply to specific commands.

For policies that can have both USING and WITH CHECK expressions (ALL and UPDATE), if no WITH CHECK expression is defined, then the USING expression will be used both to determine which rows are visible (normal USING case) and which new rows will be allowed to be added (WITH CHECK case).

If row−level security is enabled for a table, but no applicable policies exist, a “default deny” policy is assumed, so that no rows will be visible or updatable.

PARAMETERS

name
The name of the policy to be created. This must be distinct from the name of any other policy for the table.

table_name
The name (optionally schema−qualified) of the table the policy applies to.

PERMISSIVE
Specify that the policy is to be created as a permissive policy. All permissive policies which are applicable to a given query will be combined together using the Boolean “OR” operator. By creating permissive policies, administrators can add to the set of records which can be accessed. Policies are permissive by default.

RESTRICTIVE
Specify that the policy is to be created as a restrictive policy. All restrictive policies which are applicable to a given query will be combined together using the Boolean “AND” operator. By creating restrictive policies, administrators can reduce the set of records which can be accessed as all restrictive

policies must be passed for each record.

Note that there needs to be at least one permissive policy to grant access to records before restrictive policies can be usefully used to reduce that access. If only restrictive policies exist, then no records will be accessible. When a mix of permissive and restrictive policies are present, a record is only accessible if at least one of the permissive policies passes, in addition to all the restrictive policies.

command
The command to which the policy applies. Valid options are **ALL**, **SELECT**, **INSERT**, **UPDATE**, and **DELETE**. **ALL** is the default. See below for specifics regarding how these are applied.

role_name
The role(s) to which the policy is to be applied. The default is PUBLIC, which will apply the policy to all roles.

using_expression
Any SQL conditional expression (returning boolean). The conditional expression cannot contain any aggregate or window functions. This expression will be added to queries that refer to the table if row−level security is enabled. Rows for which the expression returns true will be visible. Any rows for which the expression returns false or null will not be visible to the user (in a **SELECT**), and will not be available for modification (in an **UPDATE** or **DELETE**). Typically, such rows are silently suppressed; no error is reported (but see Table 300 for exceptions).

check_expression
Any SQL conditional expression (returning boolean). The conditional expression cannot contain any aggregate or window functions. This expression will be used in **INSERT** and **UPDATE** queries against the table if row−level security is enabled. Only rows for which the expression evaluates to true will be allowed. An error will be thrown if the expression evaluates to false or null for any of the records inserted or any of the records that result from the update. Note that the *check_expression* is evaluated against the proposed new contents of the row, not the original contents.

Per−Command Policies

ALL
Using ALL for a policy means that it will apply to all commands, regardless of the type of command. If an ALL policy exists and more specific policies exist, then both the ALL policy and the more specific policy (or policies) will be applied. Additionally, ALL policies will be applied to both the selection side of a query and the modification side, using the USING expression for both cases if only a USING expression has been defined.

As an example, if an UPDATE is issued, then the ALL policy will be applicable both to what the UPDATE will be able to select as rows to be updated (applying the USING expression), and to the resulting updated rows, to check if they are permitted to be added to the table (applying the WITH CHECK expression, if defined, and the USING expression otherwise). If an **INSERT** or **UPDATE** command attempts to add rows to the table that do not pass the ALL policy's WITH CHECK expression (or its USING expression, if it does not have a WITH CHECK expression), the entire command will be aborted.

SELECT
Using SELECT for a policy means that it will apply to SELECT queries and whenever SELECT permissions are required on the relation the policy is defined for. The result is that only those records from the relation that pass the SELECT policy will be returned during a SELECT query, and that queries that require SELECT permissions, such as UPDATE, DELETE, and MERGE, will also only see those records that are allowed by the SELECT policy. A SELECT policy cannot have a WITH CHECK expression, as it only applies in cases where records are being retrieved from the relation, except as described below.

If a data−modifying query has a RETURNING clause, SELECT permissions are required on the relation, and any newly inserted or updated rows from the relation must satisfy the relation's SELECT

policies in order to be available to the RETURNING clause. If a newly inserted or updated row does not satisfy the relation's SELECT policies, an error will be thrown (inserted or updated rows to be returned are *never* silently ignored).

If an INSERT has an ON CONFLICT DO UPDATE clause, or an ON CONFLICT DO NOTHING clause with an arbiter index or constraint specification, then SELECT permissions are required on the relation, and the rows proposed for insertion are checked using the relation's SELECT policies. If a row proposed for insertion does not satisfy the relation's SELECT policies, an error is thrown (the INSERT is *never* silently avoided). In addition, if the UPDATE path is taken, the row to be updated and the new updated row are checked against the relation's SELECT policies, and an error is thrown if they are not satisfied (an auxiliary UPDATE is *never* silently avoided).

A MERGE command requires SELECT permissions on both the source and target relations, and so each relation's SELECT policies are applied before they are joined, and the MERGE actions will only see those records that are allowed by those policies. In addition, if an UPDATE action is executed, the target relation's SELECT policies are applied to the updated row, as for a standalone UPDATE, except that an error is thrown if they are not satisfied.

INSERT

Using INSERT for a policy means that it will apply to INSERT commands and MERGE commands that contain INSERT actions. Rows being inserted that do not pass this policy will result in a policy violation error, and the entire INSERT command will be aborted. An INSERT policy cannot have a USING expression, as it only applies in cases where records are being added to the relation.

Note that an INSERT with an ON CONFLICT DO NOTHING/UPDATE clause will check the INSERT policies' WITH CHECK expressions for all rows proposed for insertion, regardless of whether or not they end up being inserted.

UPDATE

Using UPDATE for a policy means that it will apply to UPDATE, SELECT FOR UPDATE, and SELECT FOR SHARE commands, as well as auxiliary ON CONFLICT DO UPDATE clauses of INSERT commands, and MERGE commands containing UPDATE actions. Since an UPDATE command involves pulling an existing record and replacing it with a new modified record, UPDATE policies accept both a USING expression and a WITH CHECK expression. The USING expression determines which records the UPDATE command will see to operate against, while the WITH CHECK expression defines which modified rows are allowed to be stored back into the relation.

Any rows whose updated values do not pass the WITH CHECK expression will cause an error, and the entire command will be aborted. If only a USING clause is specified, then that clause will be used for both USING and WITH CHECK cases.

Typically an UPDATE command also needs to read data from columns in the relation being updated (e.g., in a WHERE clause or a RETURNING clause, or in an expression on the right hand side of the SET clause). In this case, SELECT rights are also required on the relation being updated, and the appropriate SELECT or ALL policies will be applied in addition to the UPDATE policies. Thus the user must have access to the row(s) being updated through a SELECT or ALL policy in addition to being granted permission to update the row(s) via an UPDATE or ALL policy.

When an INSERT command has an auxiliary ON CONFLICT DO UPDATE clause, if the UPDATE path is taken, the row to be updated is first checked against the USING expressions of any UPDATE policies, and then the new updated row is checked against the WITH CHECK expressions. Note, however, that unlike a standalone UPDATE command, if the existing row does not pass the USING expressions, an error will be thrown (the UPDATE path will *never* be silently avoided). The same applies to an UPDATE action of a **MERGE** command.

DELETE

Using DELETE for a policy means that it will apply to DELETE commands and MERGE commands containing DELETE actions. For a DELETE command, only rows that pass this policy will be seen by the DELETE command. There can be rows that are visible through a SELECT policy that are not available for deletion, if they do not pass the USING expression for the DELETE policy. Note, however, that a DELETE action in a MERGE command will see rows that are visible through SELECT policies, and if the DELETE policy does not pass for such a row, an error will be thrown.

In most cases a DELETE command also needs to read data from columns in the relation that it is deleting from (e.g., in a WHERE clause or a RETURNING clause). In this case, SELECT rights are also required on the relation, and the appropriate SELECT or ALL policies will be applied in addition to the DELETE policies. Thus the user must have access to the row(s) being deleted through a SELECT or ALL policy in addition to being granted permission to delete the row(s) via a DELETE or ALL policy.

A DELETE policy cannot have a WITH CHECK expression, as it only applies in cases where records are being deleted from the relation, so that there is no new row to check.

Table 300 summarizes how the different types of policy apply to specific commands. In the table, “check” means that the policy expression is checked and an error is thrown if it returns false or null, whereas “filter” means that the row is silently ignored if the policy expression returns false or null.

Table 300. Policies Applied by Command Type

Command	SELECT/ALL policy	INSERT/ALL policy	UPDATE/ALL policy		DELETE/ALL policy
	USING expression	**WITH CHECK expression**	**USING expression**	**WITH CHECK expression**	**USING expression**
SELECT / COPY ... TO	Filter existing row	—	—	—	—
SELECT FOR UPDATE/SHARE	Filter existing row	—	Filter existing row	—	—
INSERT	Check new row [a]	Check new row	—	—	—
UPDATE	Filter existing row [a] & check new row [a]	—	Filter existing row	Check new row	—
DELETE	Filter existing row [a]	—	—	—	Filter existing row
INSERT ... ON CONFLICT	Check new row [b][c]	Check new row [c]	—	—	—
ON CONFLICT DO UPDATE	Check existing & new rows [d]	—	Check existing row	Check new row [d]	—
MERGE	Filter source & target rows	—	—	—	—
MERGE ... THEN INSERT	Check new row [a]	Check new row	—	—	—
MERGE ... THEN UPDATE	Check new row	—	Check existing row	Check new row	—
MERGE ... THEN DELETE	—	—	—	—	Check existing row

---
[a] If read access is required to either the existing or new row (for example, a WHERE or RETURNING clause that refers to columns from the relation). ----
[b] If an arbiter index or constraint is specified. ----
[c] Row proposed for insertion is checked regardless of whether or not a conflict occurs. ----
[d] New row of the auxiliary **UPDATE** command, which might be different from the new row of the original **INSERT** command.

Application of Multiple Policies

When multiple policies of different command types apply to the same command (for example, SELECT and UPDATE policies applied to an UPDATE command), then the user must have both types of permissions (for example, permission to select rows from the relation as well as permission to update them). Thus the expressions for one type of policy are combined with the expressions for the other type of policy using the AND operator.

When multiple policies of the same command type apply to the same command, then there must be at least one PERMISSIVE policy granting access to the relation, and all of the RESTRICTIVE policies must pass. Thus all the PERMISSIVE policy expressions are combined using OR, all the RESTRICTIVE policy expressions are combined using AND, and the results are combined using AND. If there are no PERMISSIVE policies, then access is denied.

Note that, for the purposes of combining multiple policies, ALL policies are treated as having the same type as whichever other type of policy is being applied.

For example, in an UPDATE command requiring both SELECT and UPDATE permissions, if there are

multiple applicable policies of each type, they will be combined as follows:

expression from RESTRICTIVE SELECT/ALL policy 1
AND
expression from RESTRICTIVE SELECT/ALL policy 2
AND
...
AND
(
  expression from PERMISSIVE SELECT/ALL policy 1
  OR
  expression from PERMISSIVE SELECT/ALL policy 2
  OR
  ...
)
AND
expression from RESTRICTIVE UPDATE/ALL policy 1
AND
expression from RESTRICTIVE UPDATE/ALL policy 2
AND
...
AND
(
  expression from PERMISSIVE UPDATE/ALL policy 1
  OR
  expression from PERMISSIVE UPDATE/ALL policy 2
  OR
  ...
)

NOTES

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

While policies will be applied for explicit queries against tables in the database, they are not applied when the system is performing internal referential integrity checks or validating constraints. This means there are indirect ways to determine that a given value exists. An example of this is attempting to insert a duplicate value into a column that is a primary key or has a unique constraint. If the insert fails then the user can infer that the value already exists. (This example assumes that the user is permitted by policy to insert records which they are not allowed to see.) Another example is where a user is allowed to insert into a table which references another, otherwise hidden table. Existence can be determined by the user inserting values into the referencing table, where success would indicate that the value exists in the referenced table. These issues can be addressed by carefully crafting policies to prevent users from being able to insert, delete, or update records at all which might possibly indicate a value they are not otherwise able to see, or by using generated values (e.g., surrogate keys) instead of keys with external meanings.

Generally, the system will enforce filter conditions imposed using security policies prior to qualifications that appear in user queries, in order to prevent inadvertent exposure of the protected data to user−defined functions which might not be trustworthy. However, functions and operators marked by the system (or the system administrator) as LEAKPROOF may be evaluated before policy expressions, as they are assumed to be trustworthy.

Since policy expressions are added to the user's query directly, they will be run with the rights of the user running the overall query. Therefore, users who are using a given policy must be able to access any tables or functions referenced in the expression or they will simply receive a permission denied error when attempting to query the table that has row−level security enabled. This does not change how views work, however. As with normal queries and views, permission checks and policies for the tables which are referenced by a view will use the view owner's rights and any policies which apply to the view owner,

except if the view is defined using the security_invoker option (see **CREATE VIEW**).

No separate policy exists for **MERGE**. Instead, the policies defined for **SELECT**, **INSERT**, **UPDATE**, and **DELETE** are applied while executing **MERGE**, depending on the actions that are performed.

Additional discussion and practical examples can be found in Section 5.9.

COMPATIBILITY

CREATE POLICY is a PostgreSQL extension.

SEE ALSO

ALTER POLICY (**ALTER_POLICY**(7)), DROP POLICY (**DROP_POLICY**(7)), ALTER TABLE (**ALTER_TABLE**(7))