create_view

- Description
- Parameters
- Notes
  - Updatable Views
- Examples
- Compatibility
- See Also

CREATE VIEW

CREATE VIEW

7

SQL - Language Statements

CREATE VIEW

define a new view

CREATE [OR REPLACE] [TEMP | TEMPORARY] [RECURSIVE] VIEW

name

[(

column_name

[, ...])] [WITH (

view_option_name

[=

view_option_value

] [, ...])] AS

query

[WITH [CASCADED | LOCAL] CHECK OPTION]

Description

`CREATE VIEW` defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

`CREATE OR REPLACE VIEW` is similar, but if a view of the same name already exists, it is replaced. The new query must generate the same columns that were generated by the existing view query (that is, the same column names in the same order and with the same data types), but it may add additional columns to the end of the list. The calculations giving rise to the output columns may be completely different.

If a schema name is given (for example, `CREATE VIEW myschema.myview ...`) then the view is created in the specified schema. Otherwise it is created in the current schema. Temporary views exist in a special schema, so a schema name cannot be given when creating a temporary view. The name of the view must be distinct from the name of any other relation (table, sequence, index, view, materialized view, or foreign table) in the same schema.

Parameters

`TEMPORARY` or `TEMP`
If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session. Existing permanent relations with the same name are not visible to the current session while the temporary view exists, unless they are referenced with schema-qualified names.

If any of the tables referenced by the view are temporary, the view is created as a temporary view (whether `TEMPORARY` is specified or not).

`RECURSIVE`
Creates a recursive view. The syntax CREATE RECURSIVE VIEW [<schema> .] <view_name> (<column_names>) AS SELECT <...>; is equivalent to CREATE VIEW [<schema> .] <view_name> AS WITH RECURSIVE <view_name> (<column_names>) AS (SELECT <...>) SELECT <column_names> FROM <view_name>; A view column name list must be specified for a recursive view.

<name>
The name (optionally schema-qualified) of a view to be created.

<column_name>
An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

`WITH ( view_option_name [= view_option_value] [, ... ] )`
This clause specifies optional parameters for a view; the following parameters are supported:

`check_option` (`enum`)
This parameter may be either `local` or `cascaded`, and is equivalent to specifying `WITH [ CASCADED | LOCAL ] CHECK OPTION` (see below).

`security_barrier` (`boolean`)
This should be used if the view is intended to provide row-level security. See ??? for full details.

`security_invoker` (`boolean`)
This option causes the underlying base relations to be checked against the privileges of the user of the view rather than the view owner. See the notes below for full details.

All of the above options can be changed on existing views using `ALTER VIEW`.

<query>
A `SELECT` or `VALUES` command which will provide the columns and rows of the view.

`WITH [ CASCADED | LOCAL ] CHECK OPTION`
This option controls the behavior of automatically updatable views. When this option is specified, `INSERT`, `UPDATE`, and `MERGE` commands on the view will be checked to ensure that new rows satisfy the view-defining condition (that is, the new rows are checked to ensure that they are visible through the view). If they are not, the update will be rejected. If the `CHECK OPTION` is not specified, `INSERT`, `UPDATE`, and `MERGE` commands on the view are allowed to create rows that are not visible through the view. The following check options are supported:

`LOCAL`
New rows are only checked against the conditions defined directly in the view itself. Any conditions defined on underlying base views are not checked (unless they also specify the `CHECK OPTION`).

`CASCADED`
New rows are checked against the conditions of the view and all underlying base views. If the `CHECK OPTION` is specified, and neither `LOCAL` nor `CASCADED` is specified, then `CASCADED` is assumed.

The `CHECK OPTION` may not be used with `RECURSIVE` views.

Note that the `CHECK OPTION` is only supported on views that are automatically updatable, and do not have `INSTEAD OF` triggers or `INSTEAD` rules. If an automatically updatable view is defined on top of a

base view that has `INSTEAD OF` triggers, then the `LOCAL CHECK OPTION` may be used to check the conditions on the automatically updatable view, but the conditions on the base view with `INSTEAD OF` triggers will not be checked (a cascaded check option will not cascade down to a trigger-updatable view, and any check options defined directly on a trigger-updatable view will be ignored). If the view or any of its base relations has an `INSTEAD` rule that causes the `INSERT` or `UPDATE` command to be rewritten, then all check options will be ignored in the rewritten query, including any checks from automatically updatable views defined on top of the relation with the `INSTEAD` rule. `MERGE` is not supported if the view or any of its base relations have rules.

Notes

Use the `DROP VIEW` statement to drop views.

Be careful that the names and types of the view's columns will be assigned the way you want. For example:

```
CREATE VIEW vista AS SELECT 'Hello World';
```

is bad form because the column name defaults to `?column?`; also, the column data type defaults to `text`, which might not be what you wanted. Better style for a string literal in a view's result is something like:

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

By default, access to the underlying base relations referenced in the view is determined by the permissions of the view owner. In some cases, this can be used to provide secure but restricted access to the underlying tables. However, not all views are secure against tampering; see ??? for details.

If the view has the `security_invoker` property set to `true`, access to the underlying base relations is determined by the permissions of the user executing the query, rather than the view owner. Thus, the user of a security invoker view must have the relevant permissions on the view and its underlying base relations.

If any of the underlying base relations is a security invoker view, it will be treated as if it had been accessed directly from the original query. Thus, a security invoker view will always check its underlying base relations using the permissions of the current user, even if it is accessed from a view without the `security_invoker` property.

If any of the underlying base relations has row-level security enabled, then by default, the row-level security policies of the view owner are applied, and access to any additional relations referred to by those policies is determined by the permissions of the view owner. However, if the view has

`security_invoker` set to `true`, then the policies and permissions of the invoking user are used instead, as if the base relations had been referenced directly from the query using the view.

Functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore, the user of a view must have permissions to call all functions used by the view. Functions in the view are executed with the privileges of the user executing the query or the function owner, depending on whether the functions are defined as `SECURITY INVOKER` or `SECURITY DEFINER`. Thus, for example, calling `CURRENT_USER` directly in a view will always return the invoking user, not the view owner. This is not affected by the view's `security_invoker` setting, and so a view with `security_invoker` set to `false` is *not* equivalent to a `SECURITY DEFINER` function and those concepts should not be confused.

The user creating or replacing a view must have `USAGE` privileges on any schemas referred to in the view query, in order to look up the referenced objects in those schemas. Note, however, that this lookup only happens when the view is created or replaced. Therefore, the user of the view only requires the `USAGE` privilege on the schema containing the view, not on the schemas referred to in the view query, even for a security invoker view.

When `CREATE OR REPLACE VIEW` is used on an existing view, only the view's defining SELECT rule, plus any `WITH ( ... )` parameters and its `CHECK OPTION` are changed. Other view properties, including ownership, permissions, and non-SELECT rules, remain unchanged. You must own the view to replace it (this includes being a member of the owning role).

Updatable Views

updatable views

Simple views are automatically updatable: the system will allow `INSERT`, `UPDATE`, `DELETE`, and `MERGE` statements to be used on the view in the same way as on a regular table. A view is automatically updatable if it satisfies all of the following conditions:

- The view must have exactly one entry in its `FROM` list, which must be a table or another updatable view.
- The view definition must not contain `WITH`, `DISTINCT`, `GROUP BY`, `HAVING`, `LIMIT`, or `OFFSET` clauses at the top level.
- The view definition must not contain set operations (`UNION`, `INTERSECT` or `EXCEPT`) at the top level.
- The view's select list must not contain any aggregates, window functions or set-returning functions.

An automatically updatable view may contain a mix of updatable and non-updatable columns. A column is updatable if it is a simple reference to an updatable column of the underlying base relation; otherwise the column is read-only, and an error will be raised if an `INSERT`, `UPDATE`, or `MERGE` statement attempts to assign a value to it.

If the view is automatically updatable the system will convert any `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statement on the view into the corresponding statement on the underlying base relation. `INSERT` statements that have an `ON CONFLICT` clause are fully supported.

If an automatically updatable view contains a `WHERE` condition, the condition restricts which rows of the base relation are available to be modified by `UPDATE`, `DELETE`, and `MERGE` statements on the view. However, an `UPDATE` or `MERGE` is allowed to change a row so that it no longer satisfies the `WHERE` condition, and thus is no longer visible through the view. Similarly, an `INSERT` or `MERGE` command can potentially insert base-relation rows that do not satisfy the `WHERE` condition and thus are not visible through the view (`ON CONFLICT DO SELECT/UPDATE` may similarly affect an existing row not visible through the view). The `CHECK OPTION` may be used to prevent `INSERT`, `UPDATE`, and `MERGE` commands from creating such rows that are not visible through the view.

If an automatically updatable view is marked with the `security_barrier` property then all the view's `WHERE` conditions (and any conditions using operators which are marked as `LEAKPROOF`) will always be evaluated before any conditions that a user of the view has added. See ??? for full details. Note that, due to this, rows which are not ultimately returned (because they do not pass the user's `WHERE` conditions) may still end up being locked. `EXPLAIN` can be used to see which conditions are applied at the relation level (and therefore do not lock rows) and which are not.

A more complex view that does not satisfy all these conditions is read-only by default: the system will not allow an `INSERT`, `UPDATE`, `DELETE`, or `MERGE` on the view. You can get the effect of an updatable view by creating `INSTEAD OF` triggers on the view, which must convert attempted inserts, etc. on the view into appropriate actions on other tables. For more information see ???. Another possibility is to create rules (see ???), but in practice triggers are easier to understand and use correctly. Also note that `MERGE` is not supported on relations with rules.

Note that the user performing the insert, update or delete on the view must have the corresponding insert, update or delete privilege on the view. In addition, by default, the view's owner must have the relevant privileges on the underlying base relations, whereas the user performing the update does not need any permissions on the underlying base relations (see ???). However, if the view has `security_invoker` set to `true`, the user performing the update, rather than the view owner, must have the relevant privileges on the underlying base relations.

Examples

Create a view consisting of all comedy films:

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
```

```
    WHERE kind = 'Comedy';
```

This will create a view containing the columns that are in the film table at the time of view creation. Though `*` was used to create the view, columns added later to the table will not be part of the view.

Create a view with `LOCAL CHECK OPTION`:

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

This will create a view based on the comedies view, showing only films with `kind = 'Comedy'` and `classification = 'U'`. Any attempt to `INSERT` or `UPDATE` a row in the view will be rejected if the new row doesn't have `classification = 'U'`, but the film kind will not be checked.

Create a view with `CASCADED CHECK OPTION`:

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

This will create a view that checks both the kind and classification of new rows.

Create a view with a mix of updatable and non-updatable columns:

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

This view will support `INSERT`, `UPDATE` and `DELETE`. All the columns from the films table will be updatable, whereas the computed columns country and avg_rating will be read-only.

Create a recursive view consisting of the numbers from 1 to 100:

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

Notice that although the recursive view's name is schema-qualified in this `CREATE`, its internal self-reference is not schema-qualified. This is because the implicitly-created CTE's name cannot be schema-qualified.

Compatibility

`CREATE OR REPLACE VIEW` is a PostgreSQL language extension. So is the concept of a temporary view. The `WITH ( ... )` clause is an extension as well, as are security barrier views and security invoker views.

See Also

□□□ #2
□ □□□ □□□ 2026-10-31 14:10:14 UTC
□ □□□ □□□ 2026-10-31 14:10:15 UTC