

Access Control

Row level security



Copyright

© Postgres Professional, 2015–2022

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Translated by Alexander Meleshko

Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

What is row level security

When RLS policies apply

Multiple policies, same table

Row level security policy



Determines the visibility and mutability of table rows

the predicate is calculated *for each row* using the initiating user's privileges
the client can only access the rows for which the predicate is true

Predicate for existing rows (USING)

used by SELECT, UPDATE, DELETE commands
policy violation does not trigger an error
(unless the *row_security* parameter is unset)

Predicate for new rows (WITH CHECK)

used by INSERT, UPDATE commands
if omitted, the first predicate is used
policy violation triggers an error

3

Row level security (RLS) policies allow the system administrator to control user access to a table at the level of individual rows. This mechanism is also known as Fine-Grained Access Control.

It is a supplemental tool, as the role must have necessary privileges to access the table in the first place.

Row level security policies determine if a certain user should be able to read or modify a certain row by calculating one of two predicates (binary expressions) for each of the queried rows. The predicate result determines if the user would be allowed to access the row.

The first predicate is used for *existing* rows. It is used by the commands SELECT, UPDATE, and DELETE. If the predicate for a given row does not return *true* (that is, the function returns *false* or *NULL*), the row isn't included into the client's result set. As a gross oversimplification, you can imagine that the predicate is simply appended to the WHERE clause of the query. In reality, however, it is more complicated.

If the *row_security* parameter value is set to *off*, a false predicate for even a single row will result in a query error. This is useful when making a logical backup to ensure that all rows of all tables got in.

The second predicate determines the visibility of *new* rows. It is used by the INSERT and UPDATE commands and always throws an error if the policy is violated.

<https://postgrespro.com/docs/postgresql/13/ddl-rowsecurity>

When RLS policies apply



Policy applies

- to the table for which RLS is enabled
- for specified roles and operators
(SELECT, INSERT, UPDATE, DELETE)

Policy does not apply

- during integrity constraints verification
- for superusers and roles with the BYPASSRLS attribute
- for the owner (unless enabled explicitly)

4

In order for row level security policies to start working, this mechanism must be explicitly enabled for each table.

When creating a RLS policy, you can specify what roles (by default, all) and what operators (by default, also all) will the policy apply to.

Policies are not applied during integrity constraints verification: PostgreSQL must guarantee data integrity regardless of security configurations.

Policies are not applied for superusers (as with any other security checks) and for roles with the BYPASSRLS attribute.

For the owner of the table, the policies do not apply by default, but can be enabled if necessary.

Row level security policy example

```
=> CREATE DATABASE access_rls;
```

```
CREATE DATABASE
```

```
=> \c access_rls
```

You are now connected to database "access_rls" as user "student".

Alice and Bob work in different departments of the same company.

```
student=# CREATE ROLE alice LOGIN;
```

```
CREATE ROLE
```

```
student=# CREATE ROLE bob LOGIN;
```

```
CREATE ROLE
```

```
student=# CREATE TABLE users_depts(  
    login text,  
    department text  
);
```

```
CREATE TABLE
```

```
student=# INSERT INTO users_depts VALUES ('alice','PR'), ('bob','Sales');
```

```
INSERT 0 2
```

They work with the same table that contains information from different departments. However, both Alice and Bob must only see the data from their own departments.

```
student=# CREATE TABLE revenue(  
    department text,  
    amount numeric(10,2)  
);
```

```
CREATE TABLE
```

```
student=# INSERT INTO revenue SELECT 'PR', -random()* 100.00 FROM generate_series(1,100000);
```

```
INSERT 0 100000
```

```
student=# INSERT INTO revenue SELECT 'Sales', random()*1000.00 FROM generate_series(1,10000);
```

```
INSERT 0 10000
```

Define an appropriate policy and enable it:

```
student=# CREATE POLICY departments ON revenue  
    USING (department = (SELECT department FROM users_depts WHERE login = current_user));
```

```
CREATE POLICY
```

```
student=# ALTER TABLE revenue ENABLE ROW LEVEL SECURITY;
```

```
ALTER TABLE
```

Grant Alice and Bob the privileges:

```
student=# GRANT SELECT ON users_depts, revenue TO alice, bob;
```

```
GRANT
```

The superuser (and also the owner, in this case) sees all the rows, as they ignore the policy restrictions:

```
student=# SELECT department, sum(amount) FROM revenue GROUP BY department;
```

```
department |      sum  
-----+-----  
PR          | -4995446.82  
Sales       |  5042529.14  
(2 rows)
```

What do Alice and Bob see?

```
student=# \c - alice
```

You are now connected to database "access_rls" as user "alice".

```
alice=> SELECT department, sum(amount) FROM revenue GROUP BY department;
```

department	sum
PR	-4995446.82

(1 row)

```
| => \c access_rols bob
```

```
| You are now connected to database "access_rols" as user "bob".
```

```
| bob=> SELECT department, sum(amount) FROM revenue GROUP BY department;
```

department	sum
Sales	5042529.14

(1 row)

Multiple policies

Permissive policies

visibility must be allowed by at least one permissive policy
if no policy allows visibility, the row is not visible

Restrictive policies

visibility must be allowed by all restrictive policies, unless none exist

Multiple policies can be defined on a single table. In this case, all predicates will be considered.

By default, created policies are *permissive*. For a row to be visible, *at least one* of the predicates of these policies must be *true*.

But if row level security is enabled and no permissive policy is defined, not a single row will be available.

Additionally, you can define *restrictive* policies. If such policies are defined, *all of them* must return *true* for the row to be visible.

In other words, if only permissive policies are defined and the predicates are P_1, \dots, P_N , then for each row the following expression is evaluated:

$$P_1 \text{ OR } \dots \text{ OR } P_N.$$

And if restrictive policies R_1, \dots, R_M are also defined, then the evaluated expression will be

$$(P_1 \text{ OR } \dots \text{ OR } P_N) \text{ AND } R_1 \text{ AND } \dots \text{ AND } R_M.$$

The bottom line is, visibility must be allowed *by at least one permissive policy* and *by all restrictive policies*.

Multiple policies

Allow Bob to add rows to the table, but only for his department and only under 100 USD:

- the first restriction will apply automatically (the same predicate works for both existing and newly created rows),
- the second restriction needs a new policy created for it.

```
alice=> \c - student
```

You are now connected to database "access_rls" as user "student".

```
student=# CREATE POLICY amount ON revenue AS RESTRICTIVE
        USING (true)                -- all existing rows are visible
        WITH CHECK (abs(amount) <= 100.00); -- must be true for new rows
```

```
CREATE POLICY
```

```
student=# GRANT INSERT ON revenue TO bob;
```

```
GRANT
```

Verify:

```
| bob=> INSERT INTO revenue VALUES ('Sales', 42.00);
| INSERT 0 1
| bob=> INSERT INTO revenue VALUES ('PR', 42.00);
| ERROR:  new row violates row-level security policy for table "revenue"
| bob=> INSERT INTO revenue VALUES ('Sales', 1000.00);
| ERROR:  new row violates row-level security policy "amount" for table "revenue"
```

To see what policies exist for a given object, use psql commands \d (object description) and \dp (privilege description), for example:

```
student=# \d revenue
```

Column	Type	Collation	Nullable	Default
department	text			
amount	numeric(10,2)			

Policies:

```
POLICY "amount" AS RESTRICTIVE
  USING (true)
  WITH CHECK ((abs(amount) <= 100.00))
POLICY "departments"
  USING ((department = ( SELECT users_depts.department
                        FROM users_depts
                        WHERE (users_depts.login = CURRENT_USER))))
```

This data is also available in the pg_policies view in the system catalog.

Takeaways



Privileges control access to tables and columns,
row level security policies control access to rows

Policies are easier to set up and work more efficiently than
view and trigger based implementations

1. Continuing the example from the demo, create a role for Charlie and assign him *two* departments in the user table.
2. Define row level security policies in such a way that:
 - roles can only see the rows from their departments,
 - roles associated with a single department could add rows with the amount of up to \$100,
 - roles associated with multiple departments could add rows with any amount.
3. Verify that the policies are set up correctly.
4. Estimate the overhead costs of row level security policies by running the same query as a regular user and as a superuser.

1. Roles and tables

```
=> CREATE DATABASE access_rls;

CREATE DATABASE

=> \c access_rls

You are now connected to database "access_rls" as user "student".

student=# CREATE ROLE alice LOGIN;

CREATE ROLE

student=# CREATE ROLE bob LOGIN;

CREATE ROLE

student=# CREATE ROLE charlie LOGIN;

CREATE ROLE

student=# CREATE TABLE users_depts(
    login text,
    department text
);

CREATE TABLE

student=# INSERT INTO users_depts VALUES
    ('alice', 'PR'),
    ('bob', 'Sales'),
    ('charlie', 'PR'),
    ('charlie', 'Sales');

INSERT 0 4

student=# CREATE TABLE revenue(
    department text,
    amount numeric(10,2)
);

CREATE TABLE

student=# INSERT INTO revenue SELECT 'PR', -random()* 100.00 FROM generate_series(1,100000);

INSERT 0 100000

student=# INSERT INTO revenue SELECT 'Sales', random()*1000.00 FROM generate_series(1,10000);

INSERT 0 10000
```

2. Policies and privileges

```
student=# CREATE POLICY departments ON revenue
    USING (department IN (SELECT department FROM users_depts WHERE login = current_user));

CREATE POLICY

student=# CREATE POLICY amount ON revenue AS RESTRICTIVE
    USING (true)
    WITH CHECK (
        (SELECT count(*) FROM users_depts WHERE login = current_user) > 1
        OR abs(amount) <= 100.00
    );

CREATE POLICY

student=# ALTER TABLE revenue ENABLE ROW LEVEL SECURITY;

ALTER TABLE

student=# GRANT SELECT ON users_depts TO alice, bob, charlie;

GRANT

student=# GRANT SELECT, INSERT ON revenue TO alice, bob, charlie;

GRANT
```

3. Verify

Alice:

```
student=# \c - alice
```

You are now connected to database "access_rls" as user "alice".

```
alice=> SELECT department, sum(amount) FROM revenue GROUP BY department;
```

department	sum
PR	-4991269.96

(1 row)

```
alice=> INSERT INTO revenue VALUES ('PR', 100.00);
```

INSERT 0 1

```
alice=> INSERT INTO revenue VALUES ('PR', 101.00);
```

ERROR: new row violates row-level security policy "amount" for table "revenue"

Bob:

```
alice=> \c - bob
```

You are now connected to database "access_rls" as user "bob".

```
bob=> SELECT department, sum(amount) FROM revenue GROUP BY department;
```

department	sum
Sales	4994106.05

(1 row)

```
bob=> INSERT INTO revenue VALUES ('Sales', 100.00);
```

INSERT 0 1

```
bob=> INSERT INTO revenue VALUES ('Sales', 101.00);
```

ERROR: new row violates row-level security policy "amount" for table "revenue"

Charlie:

```
bob=> \c - charlie
```

You are now connected to database "access_rls" as user "charlie".

```
charlie=> SELECT department, sum(amount) FROM revenue GROUP BY department;
```

department	sum
PR	-4991169.96
Sales	4994206.05

(2 rows)

```
charlie=> INSERT INTO revenue VALUES ('PR', 1000.00);
```

INSERT 0 1

```
charlie=> INSERT INTO revenue VALUES ('Sales', 1000.00);
```

INSERT 0 1

4. Overhead

Run the query several times to get the average execution time.

```
charlie=> \timing on
```

Timing is on.

First as charlie:

```
charlie=> SELECT department, sum(amount) FROM revenue GROUP BY department;
```

department	sum
PR	-4990169.96
Sales	4995206.05

(2 rows)

Time: 95.967 ms

charlie=> **SELECT** department, **sum**(amount) **FROM** revenue **GROUP BY** department;

department	sum
PR	-4990169.96
Sales	4995206.05

(2 rows)

Time: 100.821 ms

charlie=> **SELECT** department, **sum**(amount) **FROM** revenue **GROUP BY** department;

department	sum
PR	-4990169.96
Sales	4995206.05

(2 rows)

Time: 102.398 ms

Now do that again as the owner of the table, who is unaffected by the policies by default:

charlie=> \c - student

You are now connected to database "access_rols" as user "student".

student=# **SELECT** department, **sum**(amount) **FROM** revenue **GROUP BY** department;

department	sum
PR	-4990169.96
Sales	4995206.05

(2 rows)

Time: 73.021 ms

student=# **SELECT** department, **sum**(amount) **FROM** revenue **GROUP BY** department;

department	sum
PR	-4990169.96
Sales	4995206.05

(2 rows)

Time: 70.803 ms

student=# **SELECT** department, **sum**(amount) **FROM** revenue **GROUP BY** department;

department	sum
PR	-4990169.96
Sales	4995206.05

(2 rows)

Time: 70.908 ms

The overhead isn't dramatic in this case, but not negligible either.