Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Roles and attributes

Connecting to a server

Privileges

Row-level security policies

Roles and Attributes

A role is a database user

- roles are not associated with OS users

Role properties are defined by attributes

`LOGIN`	permission to connect
`SUPERUSER`	a superuser
`CREATEDB`	permission to create databases
`CREATEROLE`	permission to create roles
`REPLICATION`	using the replication protocol

etc.

A role is a database user. (A role can also comprise a *group* of users, but we'll discuss it later.)

Formally, roles are not associated with operating system users in any way, but many programs imply it when choosing default values. For example, if psql is started on behalf of the `student` OS user, the connection is established on behalf of the database role with the same name, i.e., `student` (unless another role is explicitly specified in the psql options).

At the time of cluster initialization, an initial role is defined, which has superuser privileges (this role is usually called postgres). Later on, you can create, modify, and delete roles.

https://postgrespro.com/docs/postgresql/12/database-roles

A role has several *attributes* that define its general properties and rights (unrelated to object access).

There are usually two flavors of each attribute; for example, `CREATEDB` (gives the right to create a database) and `NOCREATEDB` (gives no such right). As a rule, a restrictive flavor is the default one.

If a role has no `LOGIN` attribute, it cannot connect to a server. (Such roles can be used as group ones.)

This slide lists only some of the available attributes. `INHERIT` and `BYPASSRLS` attributes will be covered in more detail further in this lecture.

https://postgrespro.com/docs/postgresql/12/role-attributes

https://postgrespro.com/docs/postgresql/12/sql-createrole

Roles and Attributes

Let’s create a role for user Alice. The command specifies two attributes.

In the context of this demo, it is important to see the name of the role that executes commands, so the name of the current role is displayed in the prompt.

```
student=# CREATE ROLE alice LOGIN PASSWORD 'alicepass';
```

```
CREATE ROLE
```

The following command displays the list of roles:

```
student=# \du
```

```
                                   List of roles
 Role name |                         Attributes                         | Member of
-----------+------------------------------------------------------------+-----------
 alice     |                                                            | {}
 buyer     |                                                            | {}
 employee  |                                                            | {}
 postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 student   | Superuser                                                  | {}
```

Note that the student role is a superuser. That’s why there has been no need to take care of access rights so far.

Let’s create a database as well:

```
student=# CREATE DATABASE access_overview;
```

```
CREATE DATABASE
```

Connection

1. The rows of pg_hba.conf are searched from top to bottom
2. The first row that corresponds to the provided connection parameters (type, database, user, address) will be used

```
# TYPE          DATABASE        USER            ADDRESS                 METHOD
local           all             postgres                                peer
local           all             all                                     peer
host            all             all             127.0.0.1/32            md5
host            all             all             ::1/128                 md5
```

local — socket
host — TCP/IP

all — any database
database name

all — any role
role name

listen_addresses

all — any IP
IP/mask
domain name

For each new client, the server has to evaluate whether a database connection should be allowed. Connection parameters are defined in the pg_hba.conf configuration file (hba stands for host-based authentication). As with the main configuration file (postgresql.conf), changes come into effect only after the server reloads this file (`SELECT pg_reload_conf()` in SQL, or `pg_ctl reload` in the operating system terminal).

When a new client appears, the server reads the configuration file from top to bottom to find the row that matches the requested connection. The match is defined by four fields: connection type, database name, user name, and IP address.

Here we list only the main basic options.

Connection: `local` (unix sockets, unavailable for Windows) or `host` (a TCP/IP connection).

Database: `all` (this keyword corresponds to any database) or the name of a particular database.

User: `all` or the name of a particular role.

Address: `all`, a particular IP address with a mask, or a domain name. The address is omitted for the `local` connection type. By default, PostgreSQL listens for incoming connections only on localhost; the *listen_addresses* parameter is usually set to * (listen on all interfaces), while the access is controlled using pg_hba.conf settings.

https://postgrespro.com/docs/postgresql/12/client-authentication

Connection

3. The server performs authentication using the chosen method
4. If successful, access is allowed; otherwise, it is forbidden
   (if no rows match the given parameters, access is forbidden)

```
# TYPE          DATABASE      USER              ADDRESS             METHOD
local           all           postgres                              peer
local           all           all                                   peer
host            all           all               127.0.0.1/32        md5
host            all           all               ::1/128             md5
```

trust — allow all
reject — forbid all
scram-sha-256 and md5 — request a password
peer — ask OS

Once the server finds an appropriate row in the file, it performs client authentication using the method specified in this row, and checks for the `LOGIN` attribute and the `CONNECT` privilege. If everything is OK, the connection is allowed; otherwise, it is forbidden (other rows won't be considered in this case).

If no appropriate row is found, the access is also forbidden.

Thus, more specific connection rows should precede more generic ones while the file is viewed from top to bottom.

There are a lot of different authentication methods:

https://postgrespro.com/docs/postgresql/12/auth-methods

Here we mention only some of the main ones.

The `trust` method allows connections unconditionally. If security is not a concern, you can specify the `trust` method and use `all` for all the other parameters; then all connections will be allowed.

The `reject` method, on the contrary, unconditionally forbids all connections.

The most popular methods are `md5` and a more secure `scram-sha-256`. These methods ask for a password and check that the provided password matches the one stored in the system catalog of the database cluster.

The `peer` method checks the name of the operating system user and allows connections on behalf of the database user with the same name (you can also define a different name-mapping pattern).

Password Authentication

At the server side

- the password is set when the role is created and can be altered later
- a user that has no password won't be able to connect
- the password is stored in the `pg_authid` table of the system catalog

Entering the password on the client

- manually
- using the PGPASSWORD environment variable
- using the ~/.pgpass file (its lines have the following format: *node:port:databse:role:password*)

If password authentication is used, there must be a reference password stored for the user; otherwise the connection will be rejected.

Passwords are stored in the `pg_authid` table of the system catalog.

The user can either enter the password manually, or automate password input using one of the following options.

The first one is to define the password on the client in the PGPASSWORD environment variable. However, it is inconvenient if you have to connect to several databases, and it is not recommended for security reasons.

The second option to store passwords on the client is to use the ~/.pgpass file. The access to this file must be allowed to its owner only, otherwise PostgreSQL will ignore it.

Connection

To be able to connect to a database, a role must have the LOGIN attribute, and the pg_hba.conf file must allow connections for this role. The pg_hba.conf file can be read from SQL:

```
student=# SELECT type, database, user_name, address, auth_method
FROM pg_hba_file_rules();
```

```
 type  |   database    | user_name  |  address  | auth_method
-------+---------------+------------+-----------+-------------
 local | {all}         | {postgres} |           | peer
 local | {all}         | {all}      |           | peer
 host  | {all}         | {all}      | 127.0.0.1 | md5
 host  | {all}         | {all}      | ::1       | md5
 local | {replication} | {all}      |           | peer
 host  | {replication} | {all}      | 127.0.0.1 | md5
 host  | {replication} | {all}      | ::1       | md5
(7 rows)
```

(Depending on the PostrgeSQL distribution, the contents of the file may differ.)

We are going to use a TCP/IP host connection. This connection type corresponds to the third line in pg_hba.conf. It requires password authentication.

The alice role was created with a password, but you can change it any time:

```
student=# ALTER ROLE alice PASSWORD 'alicepass';
```

```
ALTER ROLE
```

Let's try to connect to the database by providing all the required information in the connection string:

```
student$ psql "host=localhost user=alice dbname=access_overview password=alicepass"
```

```
alice=> \conninfo
```

```
You are connected to database "access_overview" as user "alice" on host "localhost" (address "127.0.0.1") at port "5432".
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
```

Success!

Privileges

Privileges define roles' access rights for different objects

Tables

`SELECT`	read data	} can be used for columns
`INSERT`	insert rows	
`UPDATE`	update rows	
`REFERENCES`	set a foreign key	
`DELETE`	delete rows	
`TRUNCATE`	truncate a table	
`TRIGGER`	create triggers	

Views

`SELECT`	read data
`TRIGGER`	create triggers

Privileges are defined for combinations of roles and database objects. They determine the actions that roles can perform with these objects.

There are different privileges available for different object types. The privileges that can be defined for the main object types are listed on this and the next slide.

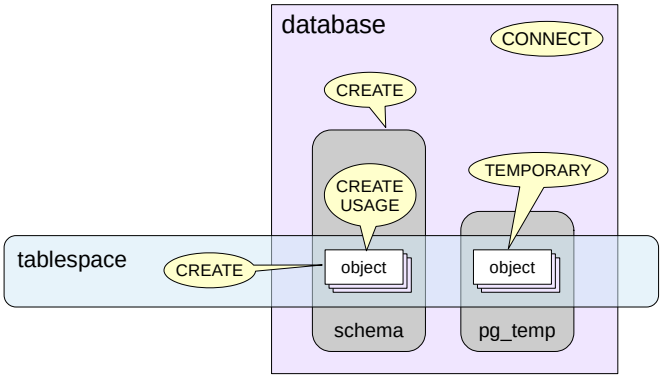
The widest choice of privileges is available for tables. Some of these privileges can be defined not only at the table level, but also at the column level.

https://postgrespro.com/docs/postgresql/12/ddl-priv

https://postgrespro.com/docs/postgresql/12/sql-grant

Privileges

Tablespaces,
databases, schemas



Sequences

SELECT	currval		
UPDATE		nextval	setval
USAGE	currval	nextval	

For sequences, the set of available privileges may seem a bit odd. Setting these privileges, you can allow or forbid access to three control functions.

For tablespaces, there is a CREATE privilege that allows creating objects in this tablespace.

When defined for a database, the CREATE privilege allows creating schemas in this database; for schemas, this privilege allows creating objects in this schema.

Since the exact name of the schema for temporary objects is unknown in advance, the privilege for creating temporary tables is defined at the database level (TEMPORARY).

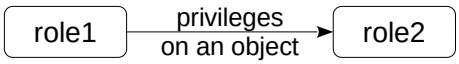
The USAGE privilege of a schema enables access to objects in this schema.

The CONNECT privilege of a database allows connections to this database.

Managing Privileges

Granting privileges

```
role1: GRANT privileges ON objects TO role2;
```



Revoking privileges

```
role1: REVOKE privileges ON object FROM role2;
```


Privileges on a particular object can be granted and revoked by the object owner (and a superuser).

The syntax of GRANT and REVOKE commands is quite complex. It allows addressing different scenarios: you can specify either some particular privileges or all the available privileges at once, grant privileges on individual objects or on groups of objects that belong to particular schemas, etc.

https://postgrespro.com/docs/postgresql/12/sql-grant

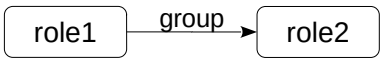
https://postgrespro.com/docs/postgresql/12/sql-revoke

Group Roles

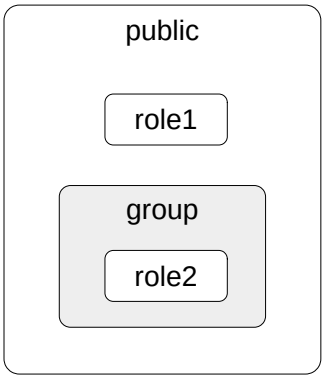
Postgres PROFESSIONAL

Including a role into a group

role1: `GRANT` *group* `TO` *role2*;



the `public` pseudorole implicitly includes all the other roles



Excluding a role from a group

role1: `REVOKE` *group* `FROM` *role2*;

Apart from representing a database user, each role can also include other roles, i.e., represent a group of roles. A role can be included into another role, just like a Unix user can be included into a group.

It is also possible to include a group role into another group (but circular membership loops are not allowed). In fact, PostgreSQL does not differentiate between single-user and group roles.

The idea of such inclusion is to make group role privileges available to single-user roles.

We can think of a group role as a predefined set of privileges that can be granted to a role just like any regular privilege. It facilitates database administration and access control.

There is also a pseudorole called `public`, which implicitly includes all the other roles. Any privilege granted to the `public` role is automatically granted to all the other roles as well.

The following roles have the right to modify a role by including or excluding other roles:

- the role to be included (or excluded)

- a role with the SUPERUSER attribute

- a role with the CREATEROLE attribute

https://postgrespro.com/docs/postgresql/12/role-membership

Superusers

This category includes

roles with the SUPERUSER attribute

Rights

unlimited access to all objects: no checks are performed

In general, we can simply say that the role's access to an object is defined by its privileges. But it makes sense to single out three categories of roles and discuss them separately.

Roles with the superuser attribute follow the most straightforward rules. Such roles bypass access control checks and can perform any operations.

Object Owner

This category includes

- initially, the role that has created the object (can be reassigned)
- any roles included into the owner role

Rights

- initially, all the privileges that can be granted on this object (can be revoked)
- actions on the owned object that are not regulated by privileges, such as deleting objects or granting and revoking privileges

Each object has an owner, i.e., the role that owns this object. Initially, it is the role that created the object, but you can change the owner later. Here is a subtle point: any role included into the owner role is also considered to be an owner.

The object owner gets the full range of privileges on this object.

In theory, these privileges can be revoked, but the object owner also has an inalienable right to perform some actions that are not regulated by privileges. In particular, the owner can grant and revoke privileges (to other roles and to itself) and delete the owned object.

Other Roles

This category includes

all other roles (which are neither superusers nor object owners)

Rights

access rights are defined by the granted privileges

group privileges are usually inherited
(but the `NOINHERIT` attribute requires the role to be used directly)

And last but not least, all the other roles can access objects as defined by the privileges granted to them. If a role belongs to a group, group-role privileges are also taken into account (in particular, those of the `public` pseudorole, which implicitly includes all the other roles).

A role usually inherits all group privileges at once. You can change this behavior by specifying the `NOINHERIT` attribute: then you'll have to explicitly call the `SET ROLE` command to use group role privileges.

To check whether a role has the required privilege on a particular object, you can call the `has_*_privilege` functions:

https://postgrespro.com/docs/postgresql/12/functions-info

It is convenient to view granted privileges using psql commands (listed in the catalogs.pdf handout).

Privileges

Alice has managed to connect to the database. Now she wants to create a separate schema and several objects in it.

```
alice=> CREATE SCHEMA alice;
```

```
ERROR:  permission denied for database access_overview
```

What has gone wrong?

Alice has no privilege to create schemas in this database. Let's grant it:

```
student=# GRANT CREATE ON DATABASE access_overview TO alice;
```

```
GRANT
```

Now try once again:

```
alice=> CREATE SCHEMA alice;
```

```
CREATE SCHEMA
```

Since Alice owns a separate schema, this role now has all the privileges for this schema and can create any objects in it. It is this schema that will be used by default:

```
alice=> SELECT current_schemas(true);
```

```
      current_schemas
---------------------------
 {pg_catalog,alice,public}
(1 row)
```

Let's create two tables.

```
alice=> CREATE TABLE t1(n numeric);
```

```
CREATE TABLE
```

```
alice=> INSERT INTO t1 VALUES (1);
```

```
INSERT 0 1
```

```
alice=> CREATE TABLE t2(n numeric, who text DEFAULT current_user);
```

```
CREATE TABLE
```

```
alice=> INSERT INTO t2(n) VALUES (1);
```

```
INSERT 0 1
```

Now let's create another role for Bob, who is going to access the objects that belong to Alice.

```
student=# CREATE ROLE bob LOGIN PASSWORD 'bobpass';
```

```
CREATE ROLE
```

```
student$ psql "host=localhost user=bob dbname=access_overview password=bobpass"
```

Bob tries to access the t1 table.

```
bob=> SELECT * FROM alice.t1;
```

```
ERROR:  permission denied for schema alice
LINE 1: SELECT * FROM alice.t1;
                      ^
```

What has caused an error?

The schema is unavailable because Bob is neither its owner nor a superuser.

To check access rights for a schema, you can run the following command (see the Access privileges column):

```
alice=> \dn+
```

```
                          List of schemas
  Name  |  Owner   |  Access privileges   |      Description
--------+----------+----------------------+------------------------
 alice  | alice    |                      |
 public | postgres | postgres=UC/postgres+| standard public schema
        |          | =UC/postgres         |
(2 rows)
```

Privileges are displayed in the following format: role=privileges/granted_by.

Each privilege is represented by one symbol; for example, schemas use the following notation:

- U = usage
- C = create

If the role name is omitted (as in the last row), the public pseudorole is implied.

If the whole field is omitted (as in the first row), the default privileges of the owner are in effect. We can see that alice has both privileges for her own schema.

Let's grant Bob access to the schema. Alice can do it as the schema owner.

```
alice=> GRANT CREATE, USAGE ON SCHEMA alice TO bob;
```

```
GRANT
```

Bob tries to access the table again:

```
bob=> SELECT * FROM alice.t1;
```

```
ERROR:  permission denied for table t1
```

What has caused an error?

Now Bob has access to the schema, but has no access to the table itself. Here is how we can check access rights:

```
alice=> \dp alice.t1
```

```
                            Access privileges
 Schema | Name | Type  | Access privileges | Column privileges | Policies
--------+------+-------+-------------------+-------------------+----------
 alice  | t1   | table |                   |                   |
(1 row)
```

We see an empty field again: only the owner (i.e., Alice) has access permissions.

Alice grants Bob the right to read and update table t1:

```
alice=> GRANT SELECT,UPDATE ON alice.t1 TO bob;
```

```
GRANT
```

For the second table, Bob gets the right to insert rows and read one of the columns:

```
alice=> GRANT SELECT(n),INSERT ON alice.t2 TO bob;
```

```
GRANT
```

Let's see how the privileges have changed:

```
alice=> \dp alice.*
```

```
                             Access privileges
 Schema | Name | Type  |  Access privileges  | Column privileges | Policies
--------+------+-------+---------------------+-------------------+----------
 alice  | t1   | table | alice=arwdDxt/alice+|                   |
        |      |       | bob=rw/alice        |                   |
 alice  | t2   | table | alice=arwdDxt/alice+| n:               +|
        |      |       | bob=a/alice         |   bob=r/alice     |
(2 rows)
```

The contents of the field has become visible; we can now see the full list of privileges in there. Here is the adopted notation, which is sometimes not quite intuitive:

- a = insert
- r = select
- w = update
- d = delete
- D = truncate
- x = reference
- t = trigger

Column privileges are displayed separately (in the column with the corresponding name).

Now Bob's attempts are successful. To avoid specifying the schema name all the time, Bob adds it to the search path.

```
bob=> SET search_path = public, alice;
SET
bob=> UPDATE t1 SET n = n + 1;
UPDATE 1
bob=> SELECT * FROM t1;
 n
---
 2
(1 row)
```

But other operations are still forbidden:

```
bob=> DELETE FROM t1;
ERROR:  permission denied for table t1
```

Bob can also access the first column of the t2 table:

```
bob=> INSERT INTO t2(n) VALUES (100);
INSERT 0 1
bob=> SELECT n FROM t2;
  n
-----
   1
 100
(2 rows)
```

But it is forbidden to read the second column:

```
bob=> SELECT * FROM t2;
ERROR:  permission denied for table t2
```

Routines

The only privilege available for functions and procedures

EXECUTE	allows routine execution

Security attributes

SECURITY INVOKER	executes a routine with the caller's rights (the default behavior)
SECURITY DEFINER	executes a routine with the owner's rights

The only privilege applicable to functions and procedures is `EXECUTE`; it allows executing the routine on which it is granted.

A subtle point is on whose behalf the routine is executed. If the routine is declared as `SECURITY INVOKER` (which is the default), it is executed with the caller's rights. In this case, the statements defined within the routine can access only those objects for which the caller has the corresponding privileges.

If the `SECURITY DEFINER` clause is specified, the routine is executed with the rights of its owner. This mechanism allows other users to perform some particular actions on the objects to which they have no direct access.

https://postgrespro.com/docs/postgresql/12/sql-createfunction

https://postgrespro.com/docs/postgresql/12/sql-createprocedure

Default Privileges

Privileges of the `public` pseudorole

- connection to any database
- access to the `public` schema and the right to create objects in it
- access to the system catalog
- execution of any routines
- privileges are automatically granted on each new object

Configuring default privileges

- a possibility to grant or revoke privileges on a newly created object

As we have already said, the `public` pseudorole includes all other roles, so they inherit all the privileges granted to `public`.

And `public` has quite an extensive list of privileges by default. In particular:

- The right to connect to any database (that's why the role `alice` could connect to the database although the `CONNECT` privilege had not been explicitly granted to this role).

- Access to the system catalog and the public schema.

- The right to execute any routines.

On the one hand, it enables seamless operation without having to deal with privileges; but on the other hand, it brings extra complications if access control is really required.

The public role automatically receives all the privileges listed above for all newly created objects. So it is not enough to simply revoke the `EXECUTE` privilege from `public`: once a new routine appears, `public` immediately gets the right to execute it.

There is a special mechanism of *default privileges* that enables you to automatically grant the required privileges on newly created objects. It can be also used to revoke the `EXECUTE` privilege from the `public` pseudorole.

https://postgrespro.com/docs/postgresql/12/sql-alterdefaultprivileges

Default Privileges and Routines

Alice creates a function:

```
alice=> CREATE FUNCTION foo() RETURNS SETOF t2
AS $$
SELECT * FROM t2;
$$ LANGUAGE sql STABLE;
```

```
CREATE FUNCTION
```

Can Bob call it if Alice has not granted him the EXECUTE privilege?

```
    bob=> SELECT foo();
```

```
    ERROR:  permission denied for table t2
    CONTEXT:  SQL function "foo" statement 1
```

Yes, the call is possible, but Bob won't be able to access the objects for which no privileges have been granted to him.

But if Bob creates table t2 in the public schema, the function will work for both users (it will use different tables since Alice and Bob have different search paths):

```
    bob=> CREATE TABLE t2(n numeric, who text DEFAULT current_user);
```

```
    CREATE TABLE
```

```
    bob=> INSERT INTO t2(n) VALUES (42);
```

```
    INSERT 0 1
```

```
    bob=> SELECT foo();
```

```
       foo
    ----------
     (42,bob)
    (1 row)
```

```
alice=> SELECT foo();
```

```
    foo
-----------
 (1,alice)
 (100,bob)
(2 rows)
```

Another possible option is to declare a function with the owner's privileges (SECURITY DEFINER):

```
alice=> ALTER FUNCTION foo() SECURITY DEFINER;
```

```
ALTER FUNCTION
```

In this case, the function is run with the privileges of the role that owns it, regardless of the identity of the caller.

Bob deletes the owned table...

```
    bob=> DROP TABLE t2;
```

```
    DROP TABLE
```

...and gets access to the contents of the table that belongs to Alice:

```
    bob=> SELECT foo();
```

```
        foo
    -----------
     (1,alice)
     (100,bob)
    (2 rows)
```

In this case, Alice should be more careful with granting privileges. It is highly likely that the EXECUTE privilege has to be revoked from the pubic role and explicitly granted only to those roles that really need it.

```
alice=> REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA alice FROM public;
```

```
REVOKE
```

```
    bob=> SELECT foo();
```

```
ERROR:  permission denied for function foo
```

What's worse, the EXECUTE privilege is automatically granted to the public role for each newly created function.

```
alice=> CREATE FUNCTION bar() RETURNS integer AS $$
SELECT 1;
$$ LANGUAGE sql IMMUTABLE SECURITY DEFINER;
```

```
CREATE FUNCTION
```

```
bob=> SELECT bar();
```

```
 bar
-----
   1
(1 row)
```

But it can be avoided if you modify the default privileges:

```
alice=> ALTER DEFAULT PRIVILEGES
FOR ROLE alice
REVOKE EXECUTE ON FUNCTIONS FROM public;
```

```
ALTER DEFAULT PRIVILEGES
```

```
alice=> \ddp
```

```
           Default access privileges
 Owner | Schema |   Type   | Access privileges
-------+--------+----------+-------------------
 alice |        | function | alice=X/alice
(1 row)
```

Now the public role has no EXECUTE privilege for functions owned by Alice, so only Alice can run them.

```
alice=> CREATE FUNCTION baz() RETURNS integer AS $$
SELECT 1;
$$ LANGUAGE sql IMMUTABLE SECURITY DEFINER;
```

```
CREATE FUNCTION
```

```
bob=> SELECT baz();
```

```
ERROR:  permission denied for function baz
```

Row-Level Security

Policies complement privileges
to manage access to tables at the row level

A policy is applied

- to particular roles
- to particular commands (`SELECT`, `INSERT`, `UPDATE`, `DELETE`)

A policy defines row access conditions

- permissive: allows row access if the condition is true
- restrictive: forbids row access if the condition is false
- different conditions (predicates) for existing rows and the rows to be inserted

While privileges provide access control at the table and column levels, row-level security policies do it at the row level.

By default, RLS is switched off. If required, it has to be enabled for each table explicitly.

Policies can be defined for a table, a set of commands (`SELECT`, `INSERT`, `UPDATE`, `DELETE`), and for particular roles. In fact, each policy is a boolean condition (a predicate), which is computed for each selected row. If the condition is true, the access is allowed (the access must not be forbidden by any *restrictive* policy and must be allowed by at least one *permissive* policy at the same time). Otherwise, the row won't be visible.

The predicates defining access to existing and newly added rows can differ (in this case, an `UPDATE` operation will be successful only if both predicates are true).

Predicates are computed with the rights of the caller.

RLS policies do not apply to the table owner (in most cases), superusers, roles with the `BYPASSRLS` attribute, and cannot be used for integrity constraints (uniqueness, foreign keys).

https://postgrespro.com/docs/postgresql/12/ddl-rowsecurity

Row-Level Security

Security policies enable us to control table access at the row level, for each particular role.

```
alice=> SELECT * FROM t2;
```

```
  n  |  who
-----+-------
   1 | alice
 100 | bob
(2 rows)
```

To see how it works, let's make the role reading the table see only its own rows, i.e., in which the who field contains this role's name.

```
alice=> CREATE POLICY who_policy ON t2
USING (who = current_user);
```

```
CREATE POLICY
```

For security policy to come into effect, it has to be enabled at the table level:

```
alice=> ALTER TABLE t2 ENABLE ROW LEVEL SECURITY;
```

```
ALTER TABLE
```

Now Bob sees only its own rows. In fact, each row has to be checked separately during query execution to verify that it satisfies the predicate specified in the policy.

```
bob=> SELECT n FROM t2;
```

```
  n
-----
 100
(1 row)
```

```
bob=> INSERT INTO t2(n) VALUES (101);
```

```
INSERT 0 1
```

```
bob=> SELECT n FROM t2;
```

```
  n
-----
 100
 101
(2 rows)
```

Row-level security policies do not apply to superusers and roles with the BYPASSRLS attribute. The table owner usually bypasses these policies as well:

```
alice=> SELECT * FROM t2;
```

```
  n  |  who
-----+-------
   1 | alice
 100 | bob
 101 | bob
(3 rows)
```

But the owner can choose to limit its own rights:

```
alice=> ALTER TABLE t2 FORCE ROW LEVEL SECURITY;
```

```
ALTER TABLE
```

```
alice=> SELECT * FROM t2;
```

```
 n |  who
---+-------
 1 | alice
(1 row)
```

Summary

Roles, privileges, and policies provide a flexible access control mechanism for different usage scenarios

- you can easily allow everything to everyone
- you can set up strict access control if required

When creating a new role, it is important to ensure that it can connect to the server

Practice

1. Create two roles (the password must match the role's name):
   – `employee`: the store's employee
   – `buyer`: a customer

   Make sure that the created roles can connect to the database.
2. Revoke the privileges to execute all functions and to connect to the database from the `public` role.
3. Configure access control as follows:
   – `employee` can only order books or add new authors and books
   – `buyer` can only purchase books

   Check that the application works as expected with these settings enabled.

Task 1. The `employee` role is an internal user of the application; its authentication is performed at the server level.

The `buyer` role is an external user. In a real online store, such users are managed by the application while all the queries are sent to the server on behalf of a single generic role (`buyer`); the ID of a particular customer can be passed as a parameter (but we do not do it in our application).

Task 3. In general, access control must be implemented in the application as well. In our bookstore application, there is no access control for a reason: instead, its web interface allows you to explicitly select the role on behalf of which to execute the query. As a result, you can see what happens on the server side if the application behavior is incorrect.

So, we need to grant the following privileges:

- The right to connect to the `bookstore` database and to access the `bookstore` schema.

- Access to views that will be called directly.

- Access to functions called as part of the API. With the default `SECURITY INVOKER` rights, the function would require access to all the underlying objects (tables and other functions). But it is more convenient to simply declare API functions as `SECURITY DEFINER`.

Naturally, the roles must be granted privileges only on those objects that they need to access.

Task 1. Creating Roles

```
=> CREATE ROLE employee LOGIN PASSWORD 'employee';
```

```
CREATE ROLE
```

```
=> CREATE ROLE buyer LOGIN PASSWORD 'buyer';
```

```
CREATE ROLE
```

The default settings allow connections from the local host using password authentication. That will do for our purposes.

Task 2. Privileges of the public Role

We need to revoke excessive privileges from the public role.

```
=> REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA bookstore FROM public;
```

```
REVOKE
```

```
=> REVOKE CONNECT ON DATABASE bookstore FROM public;
```

```
REVOKE
```

Task 3. Access Control

Functions with the owner's privileges:

```
=> ALTER FUNCTION get_catalog(text,text,boolean) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION update_catalog() SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION add_author(text,text,text) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION add_book(text,integer[]) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION buy_book(integer) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION book_name(integer,text,integer) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

```
=> ALTER FUNCTION authors(books) SECURITY DEFINER;
```

```
ALTER FUNCTION
```

The buyer's privileges: the buyer role must have access to book search and purchase functionality.

```
=> GRANT CONNECT ON DATABASE bookstore TO buyer;
```

```
GRANT
```

```
=> GRANT USAGE ON SCHEMA bookstore TO buyer;
```

```
GRANT
```

```
=> GRANT EXECUTE ON FUNCTION get_catalog(text,text,boolean) TO buyer;
```

```
GRANT
```

```
=> GRANT EXECUTE ON FUNCTION buy_book(integer) TO buyer;
```

```
GRANT
```

The employee's privileges: the employee role must have access to viewing and adding books and authors, as well as to the book catalog for ordering books.

```
=> GRANT CONNECT ON DATABASE bookstore TO employee;
```

```
GRANT
```

```
=> GRANT USAGE ON SCHEMA bookstore TO employee;
```

```
GRANT
=> GRANT SELECT,UPDATE(onhand_qty) ON catalog_v TO employee;
GRANT
=> GRANT SELECT ON authors_v TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION book_name(integer,text,integer) TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION authors(books) TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION author_name(text,text,text) TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION add_book(text,integer[]) TO employee;
GRANT
=> GRANT EXECUTE ON FUNCTION add_author(text,text,text) TO employee;
GRANT
```

Practice

Routines executed with the owner's rights (declared with the `SECURITY DEFINER` attribute) can be used to allow regular users to perform some actions that require superuser privileges.

1. Create a regular unprivileged user and check that this user cannot modify the *log_statement* parameter.
2. Implement a routine that enables and disables tracing of SQL queries so that it can be used by the created user.

Task 1. Recall the demo provided for the "PL/pgSQL. Debugging" lecture. We did not have any issues with setting this parameter because the demo was performed on behalf of the `student` user, which had superuser rights.

Task 1. Role Creation and Validation

```
student=# CREATE DATABASE access_overview;
```

```
CREATE DATABASE
```

```
student=# \c access_overview
```

```
You are now connected to database "access_overview" as user "student".
```

```
student=# CREATE ROLE alice LOGIN PASSWORD 'alicepass';
```

```
CREATE ROLE
```

```
student$ psql "host=localhost user=alice dbname=access_overview password=alicepass"
```

Alice cannot modify the following parameter value:

```
alice=> SET log_statement = 'all';
```

```
ERROR:  permission denied to set parameter "log_statement"
```

Task 2. A Tracing Procedure

On behalf of a superuser, create a procedure for modifying the parameter and make it a SECURITY DEFINER:

```
student=# CREATE PROCEDURE trace(val boolean)
AS $$
SELECT set_config(
    'log_statement',
    CASE WHEN val THEN 'all' ELSE 'none' END,
    false /* is_local */
);
$$ LANGUAGE sql SECURITY DEFINER;
```

```
CREATE PROCEDURE
```

Revoke the execution privilege from the public role...

```
student=# REVOKE EXECUTE ON PROCEDURE trace FROM public;
```

```
REVOKE
```

...and grant it to Alice. Instead of choosing between FUNCTION and PROCEDURE, almost all commands (except for CREATE) allow using ROUTINE as a generic term:

```
student=# GRANT EXECUTE ON ROUTINE trace TO alice;
```

```
GRANT
```

Now Alice can enable and disable tracing, even having no direct access to this parameter:

```
alice=> CALL trace(true);
```

```
CALL
```

```
alice=> SELECT 2*2;
```

```
 ?column?
----------
        4
(1 row)
```

```
alice=> CALL trace(false);
```

```
CALL
```

```
student$ tail -n 2 /var/log/postgresql/postgresql-12-main.log
```

```
2021-10-19 17:06:25.984 MSK [46883] alice@access_overview LOG:  statement: SELECT 2*2;
2021-10-19 17:06:26.025 MSK [46883] alice@access_overview LOG:  statement: CALL trace(false);
```