

Access Control Overview



Copyright

© Postgres Professional, 2017, 2018, 2019.
Authors: Egor Rogov, Pavel Luzanov

Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is permitted without restrictions. Commercial use is possible only with the written permission of Postgres Professional. Changes to course materials are prohibited.

Feedback

Send feedback, comments and suggestions to:
edu@postgrespro.ru

Denial of responsibility

In no event shall Postgres Professional be liable to any party for direct, indirect, special, incidental, or consequential damages, including lost profit, arising out of the use of course materials. Postgres Professional disclaims any warranties on course materials. Course materials are provided on an “as is” basis and Postgres Professional has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Roles and privileges: role-based access control

Row security policies

Authentication

Role as a user

Role attributes

Privileges

Role as a group of users

Superusers, owners, and other roles

Default privileges

The role in PostgreSQL combines two concepts: a DBMS user and a group of users.

In the past, PostgreSQL had separate entities for these concepts, and this is still reflected, for example, in the names of some commands (`CREATE USER` and `CREATE ROLE`).

Privileges define access rights of roles to database objects.

Roles and privileges constitutes the Role-Based Access Control.

Role

DBMS user (not related to OS user)

Attributes define role properties and abilities

LOGIN	connect to the server
SUPERUSER	override all access restrictions
CREATEDB	create databases
CREATEROLE	create roles
REPLICATION	use replication protocol
and some others	

A role can be considered as a DBMS user. DBMS users and OS users are different entities, although by default PostgreSQL utilities choose OS username as DBMS username for convenience. For example, when running `psql` with no parameters on behalf of the `student` OS user, `-U student` is assumed.

A role can possess some attributes that define its common properties and abilities (not related to object access rights).

Usually attributes have two options, for example, `CREATEDB` (gives the right to create a database) and `NOCREATEDB` (does not give such right). As a rule, the default is to choose the limiting option.

The table lists only some of the attributes. The attributes `INHERIT` and `BYPASSRLS` are discussed a little further in this lesson.

When initializing a cluster, one initial user is created that has superuser access. Then other roles can be created, altered, dropped.

<https://postgrespro.com/docs/postgresql/11/user-manag.html>

<https://postgrespro.com/docs/postgresql/11/database-roles.html>

<https://postgrespro.com/docs/postgresql/11/role-attributes.html>

Tables

SELECT	read rows	} possible at the column level
INSERT	insert rows	
UPDATE	change rows	
REFERENCES	reference in foreign keys	
DELETE	delete rows	
TRUNCATE	truncate table	
TRIGGER	create triggers	

Views — SELECT и TRIGGER

Sequences

SELECT	currval		
UPDATE		nextval	setval
USAGE	currval	nextval	

Privileges are defined at the intersection of objects and users. They restrict the actions available to roles for these objects.

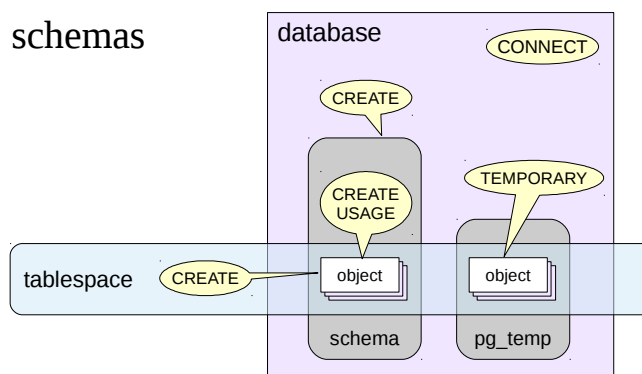
The list of possible privileges differs for objects of different types. Privileges for the most commonly used objects are shown on this and the next slide.

Tables have the most wide specter of privileges. Some of them can be defined not only for the entire table, but also for individual columns.

<https://postgrespro.com/docs/postgresql/11/sql-grant.html>

Privileges

Tablespaces, databases, schemas



Functions

EXECUTE

access depends on the definition of a function:
SECURITY INVOKER — caller rights (by default),
SECURITY DEFINER — creator rights

For tablespaces, there is the CREATE privilege that allows the creation of objects in this tablespace.

For databases, the CREATE privilege allows to create schemas in this database, and for a schema, the CREATE privilege allows to create objects in this schema.

Since the exact name of the schema for temporary objects is not known in advance, the privilege to create temporary tables has been moved to the database level (TEMPORARY).

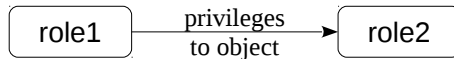
The USAGE privilege of a schema allows access to objects in this schema.

Database CONNECT privilege allows connection to this database.

For functions, there is the single EXECUTE privilege that permits the execution of this function. The subtle point is the privileges with which the function will be executed. There are two options. If the function was created as SECURITY INVOKER (default), it runs with the privileges of the calling user. If the function was created as SECURITY DEFINER, the function runs with the privileges of the user who created it.

Grant privileges

```
role1: GRANT privilege ON object TO role2;
```



Revoke privileges

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

For the user to have access to the object, an appropriate privilege must be granted to that user.

The syntax of the GRANT and REVOKE commands is quite complex and allows to specify both individual and all possible privileges; both individual objects and groups of objects included in certain schemas, etc.

<https://postgrespro.com/docs/postgresql/11/sql-grant.html>

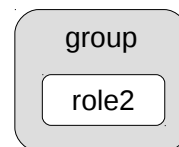
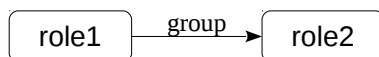
<https://postgrespro.com/docs/postgresql/11/sql-revoke.html>

Role

- may include other roles, i. e. be a «group role»
- public implicitly includes all other roles
- group privileges are inherited by membership users

Grant membership in a group

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



Revoke membership in a group

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

A role can be a member of another role just like a Unix user can be a member of a group.

The point of membership is that privileges (and attributes) of the group role become available to its members. This greatly simplifies management of privileges: you can grant necessary privileges to a group role, and then grant this role to other DBMS users as needed.

There is also the `public` pseudorole, in which all other roles are always implicitly included. This can be used to grant privileges to all users at once.

Note that PostgreSQL does not distinguish between user roles and group roles. Therefore, any role can be a member of any other. Multilevel membership is possible, but cycles are not allowed (roles cannot be members of each other).

<https://postgrespro.com/docs/postgresql/11/role-membership.html>

Who is in the category

roles with the SUPERUSER attribute

Rights

full access to all objects, no access restrictions applied

In general, it can be said that role access to an object is determined by privileges. But it makes sense to distinguish three categories of roles and consider them separately.

The easiest way is with roles with the superuser attribute. Such roles can do anything, for them access control checks are not performed.

Who is in the category

- initially the role that created the object (can be changed)
- also members of the owner role

Rights

- initially all privileges for the object (can be revoked)
- actions with own objects that are not regulated by privileges, for example: drop, grant and revoke, etc.

Each object has a role that owns this object (the owner). Initially, this is the role that created the object, although then the owner can be changed.

A non-obvious point: member of the owner role are also considered owners.

The owner of the object immediately receives the full set of privileges for this object.

In principle, these privileges can be revoked, but the owner of the object also has the imprescriptible right to take actions that are not regulated by the privileges. In particular, the owner can grant and revoke privileges to other roles (including itself), drop an object, etc.

Who is in the category

all others roles (neither superusers nor owners)

Rights

access is restricted to the granted privileges

inherit privileges of the group roles

(NOINHERIT attribute requires explicit role switching)

Finally, access of all other roles is restricted by the privileges granted to them. For members of group roles this includes the privileges of the group roles. This means that the privileges can be granted to the group as a whole. Note that all roles are implicit members of the `public` pseudorole, and so inherit all privileges of this role.

Usually the role immediately has all the group privileges. This behavior can be changed by specifying the `NOINHERIT` attribute. In this case in order to use the privileges of the group role, you will need to explicitly choose the desired role by the `SET ROLE` command.

<https://postgrespro.com/docs/postgresql/11/ddl-priv.html>

To check whether the role has the necessary privilege to access some object, you can use the functions `has_*_privilege`:

<https://postgrespro.com/docs/postgresql/11/functions-info.html>

Granted privileges are conveniently shown by `psql` commands that describe the object.

Public privileges

- connect to any database
- access the public schema and create objects in it
- access to the system catalog
- execute any functions
- privileges are granted automatically for each new object

Customizable default privileges

- automatically grant or revoke privileges for the newly created objects

The `public` pseudorole has a fairly wide range of default privileges. In particular, the right to connect to any database, access to the system catalog and the `public` schema, the right to execute any functions. Moreover, these privileges appear automatically when creating new objects. For example, as soon as a new function is created, the `public` role is immediately granted the privilege to execute it.

This allows to work comfortably without thinking about privileges, but on the other hand, it creates certain difficulties if strict access control is really necessary.

There is also a mechanism of default privileges that allows to automatically grant the necessary privileges when creating a new object. This mechanism can also be used to revoke the right to perform functions from the `public`.

<https://postgrespro.com/docs/postgresql/11/sql-alterdefaultprivileges>

In addition to the privilege system
for restricting access to tables at the row level

Privileges allow the access control at the table and column levels. In addition to this privileges system, the row-level security policies control access at the row level. This mechanism appeared in PostgreSQL 9.5.

Must be explicitly enabled for a table

- does not affect the owner (if not forced)
- does not affect roles with the BYPASSRLS attribute
- does not affect integrity constraints

Policy determines row visibility

- predicates for existing row and for new rows
- predicates are evaluated with invoker rights
- policy may be enabled for certain roles and SQL commands (SELECT, INSERT, UPDATE, DELETE)
- access is possible if it is allowed by at least one *permissive* policy and is not denied by any *restrictive* policy

Row-level access control is disabled by default. If necessary, it should be enabled explicitly for each table.

Row security policies do not affect the table owner (as a rule), roles with the special BYPASSRLS attribute, and integrity constraints (uniqueness, foreign keys). Policies are defined for a table and can be restricted to certain roles and set of SQL commands (SELECT, INSERT, UPDATE, DELETE).

In essence, each of the policies is a predicate calculated for each row of the result set. If the predicate is true, the policy is considered to allow access the row.

By default, the *permissive* policies are used. If at least one policy allows access, then the row is visible to the user. In addition to permissive policies, some *restrictive* policies can also be defined since PostgreSQL 10. All such policies must allow access for the row to be visible.

In a policy you can specify different predicates for accessing existing rows and for adding new rows (an UPDATE operation will work only if both predicates are true). While policies for the existing rows determines visibility, the policies for the new rows may lead to an error if a row is not approved.

<https://postgrespro.com/docs/postgresql/11/ddl-rowsecurity>

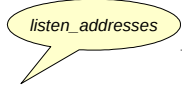
When a new client connects,
the server decides whether to allow the connection

Before controlling access to objects inside the database, for each new client the server must determine whether to allow connection to the database at all. This is called authentication.

Upon connection

1. The `pg_hba.conf` records are searched from top to bottom
2. The first record matching the connection parameters (type, base, user and address) is used for authentication

#	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		all — any role		
	host — TCP/IP		role name		
		all — any DB DB name		all — any IP IP/mask domain name	



16

Authentication settings are defined in the `pg_hba.conf` configuration file (stands for host-based authentication). Like the main configuration file `postgresql.conf`, the changes should be re-read by the server to take effect (`select pg_reload_conf()` in SQL, or `pg_ctl reload` from the operating system).

When a client appears, the postmaster spawns a new backend which performs authentication. The configuration file is searched from top to bottom for the record matching the connection type, database name, user name, and IP address.

Below are some of the most commonly used parameters, read more in: <https://postgrespro.com/docs/postgresql/11/client-authentication.html>

Connection: `local` (unix-domain sockets, not available in Windows) or `host` (TCP/IP connection).

Database: `all` matches any database, or the name of a specific database.

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

Address: `all`, specific IP address with network mask or domain name. Not specified for the `local` connections.

By default, PostgreSQL listens for incoming connections only from localhost; usually, the `listen_addresses` parameter should be set to "*" (listen to all interfaces) and access is further restricted using `pg_hba.conf`.

Upon connection

3. Authentication is performed using the chosen method and the CONNECT privilege is checked
4. Authentication succeeded — allow connection, failed — deny (when no matching record found, connection is denied)

#	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 — always allow
reject — always deny
md5, scram-sha-256 — request a password
peer — ask OS

When a matching record is found in the file, the authentication is performed using the method specified in this record. Also the CONNECT privilege is checked. If authentication succeeded, the connection is allowed, otherwise it is denied (and other records are no longer considered).

If none of the records match, then access is also denied.

Thus, the records in the file should go from top to bottom from specific to more general.

There are many authentication methods:

<https://postgrespro.com/docs/postgresql/11/auth-methods.html>. Below are only the most basic ones.

The trust method unconditionally allows the connection. If security issues are not important, you can specify "all all" and the trust method to allow all connections.

The reject method, on the contrary, absolutely prohibits the connection. It can be used for example to deny connections from specific hosts.

Probably the most common is the md5 method, which prompts the user for a password and checks if it matches the password stored on the server (in the system catalog). Starting from PostgreSQL 10 more secure method scram-sha-256 can also be used for password authentication.

The peer method requests the user name from the operating system and allows the connection if the names of the OS user and the database user are the same. You can also define other name mappings if you wish.

On server

password is set when creating or altering the role:

```
ALTER|CREATE ROLE PASSWORD 'password' VALID UNTIL 'time'
```

user without a password will not be authenticated

password is stored in the `pg_authid` system catalog

On client

enter password manually when prompted

take password from the `PGPASSWORD` environment variable

take password from the `~/.pgpass` file (*host:port:database:role:password*)

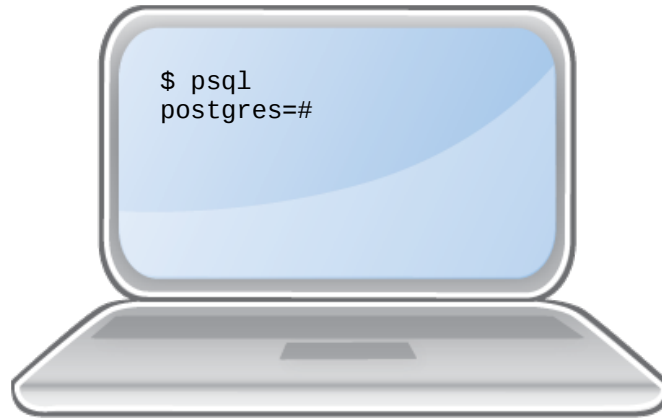
If password authentication is used, a password must be specified when creating a role, otherwise access will be denied.

The password is stored in the system catalog in the `pg_authid` table.

The user can enter the password manually when prompted by the server, or can automate the input. There are two possibilities for this.

First, the password can be specified in the `PGPASSWORD` environment variable on the client. However, this is inconvenient when accessing several databases, and is not recommended for security reasons.

Secondly, the passwords for several databases can be specified in the `~/.pgpass` file on the client. Only the owner should have access to the file, otherwise PostgreSQL will ignore it.



Roles, privileges, and row security policies provides many ways to restrict access

- everyone-can-do-everything is easy

- fine-grained access control is possible if necessary

Authentication is set up separately

1. Create a database, a schema, and a table with two columns: a key and a value.
2. Create a role.
3. Find out the IP address of the virtual machine and configure the system so that the connection from this address is allowed only to the created role and only to the created database, using password authentication.
4. Configure access control so that the created role can query the table and change the values in it, but not the keys.

3. The IP address can be found with the `ifconfig` command.