

Access Control Roles and attributes



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.

Roles

Attributes

Group role membership

Object owners

A role is a DBMS user

A role is in no way associated with the OS user

although many programs use the OS user name
as the default role name

Roles are defined at the cluster level

Essentially, a role is a DBMS user. (A role can also act as a user group, as discussed later in this topic.)

Roles have nothing to do with OS user names, although many stock programs assume the OS user name as the default role name. For example, if you do not specify the role name when starting psql, your OS user name will be used.

Roles are shared cluster objects. As such, one role can connect to different databases and own objects in different databases.

<https://postgrespro.com/docs/postgresql/13/database-roles>

Attributes define the properties of a role

```
CREATE ROLE role [WITH] attribute [attribute ...]
```

LOGIN	can log in
SUPERUSER	superuser privileges
CREATEDB	can create databases
CREATEROLE	can create roles
REPLICATION	can use the replication protocol
and others	

A role possesses a number of attributes that define its general properties and rights (not related to object access rights).

Generally, attributes come in two opposite variations, for example, CREATEDB (can create databases) and NOCREATEDB (not allowed to create databases). Usually, the restrictive option is the default.

The table lists only some of the possible attributes. The INHERIT and BYPASSRLS attributes are discussed later in this module.

<https://postgrespro.com/docs/postgresql/13/role-attributes>

<https://postgrespro.com/docs/postgresql/13/sql-createrole>

Roles and attributes

In this module, the prompt will show the name of the user that executes the command.

```
student=# CREATE DATABASE access_roles;
```

```
CREATE DATABASE
```

```
student=# \c access_roles
```

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

Create a role for Alice:

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

```
CREATE ROLE
```

Alice can log in (the LOGIN attribute) and create other roles (the CREATEROLE attribute).

Проверим это:

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

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

```
alice=> CREATE ROLE bob LOGIN;
```

```
CREATE ROLE
```

Indeed, Alice can log in and create a role for Bob.

Bob cannot create other roles:

```
student$ psql -U bob -d access_roles
```

```
| bob=> CREATE ROLE charlie LOGIN;
```

```
| ERROR: permission denied to create role
```

You can view all roles within the cluster with the following command:

```
alice=> \du
```

List of roles		
Role name	Attributes	Member of
alice	Create role	{}
bob		{}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
student	Superuser, Create role, Create DB, Replication, Bypass RLS	{}

In addition to the newly created roles alice and bob, there are two others, both with superuser privileges:

- postgres — a superuser created upon cluster initialization,
- student — a role created specifically for the course, allows us to skip providing connection parameters when using psql.

Existing roles can be modified. For example, Alice can revoke the right to log in from Bob:

```
alice=> ALTER ROLE bob NOLOGIN;
```

```
ALTER ROLE
```

Now, Bob cannot log in:

```
| bob=> \c - bob
```

```
| \connect: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed: FATAL: role "bob" is not permitted to log in
```

Alice can revoke CREATEROLE from herself:

```
alice=> ALTER ROLE alice NOCREATEROLE;
```

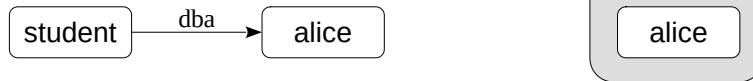
```
ALTER ROLE
```

Many attributes come in pairs, such as LOGIN-NOLOGIN and CREATEROLE-NOCREATEROLE.

Group roles

Granting group membership to a role

```
student=> GRANT dba TO alice;
```



Revoking group membership from a role

```
student=> REVOKE dba FROM alice;
```

Group role membership control

any role can *grant membership in itself* to another role

a role with the SUPERUSER attribute can grant membership in any role to any role

a role with the CREATEROLE attribute can grant membership in any non-superuser role to any role

6

A role can be granted membership in another role, just as a Unix user can be included in a group.

PostgreSQL does not distinguish between user roles and group roles, allowing any role to be a member of any other. Cascading grants may occur, unless they result in a cycle.

When a role is granted membership in another role, it obtains access to all the attributes (and privileges, more on them later) of the group role. The inclusion done with the GRANT command.

The role that executes the GRANT command is paramount. The roles that may grant (or revoke) membership in a given role are:

- the role itself,
- roles with the SUPERUSER attribute,
- roles with the CREATEROLE attribute (as long as the given role is not a superuser).

To take advantage of the newly acquired properties, you must first switch to the role by using the SET ROLE command.

<https://postgrespro.com/docs/postgresql/13/role-membership>

Group roles

Alice revoked the CREATEROLE attribute from herself and now can neither create new roles nor modify existing ones:

```
alice=> ALTER ROLE bob LOGIN;
```

```
ERROR: permission denied
```

To grant Alice superuser powers, we can include her role into student. It can be done by student or by another superuser role:

```
alice=> \c - postgres
```

You are now connected to database "access_roles" as user "postgres".

```
postgres=# GRANT student TO alice;
```

```
GRANT ROLE
```

```
postgres=# \du
```

List of roles		
Role name	Attributes	Member of
alice		{student}
bob	Cannot login	{}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
student	Superuser, Create role, Create DB, Replication, Bypass RLS	{}

Note that the LOGIN attribute isn't listed in the \du output, but its absence is.

To make sure that Alice does not abuse her superuser powers, make all her commands get recorded into the server log. To do that, we will use another method of assigning configuration parameters. This way, the new parameter value will apply when the user connects to the server:

```
postgres=# ALTER ROLE alice SET log_min_duration_statement=0;
```

```
ALTER ROLE
```

The parameter scope can be limited to a specific database:

```
postgres=# ALTER ROLE alice RESET log_min_duration_statement;
```

```
ALTER ROLE
```

```
postgres=# ALTER ROLE alice IN DATABASE access_roles SET log_min_duration_statement=0;
```

```
ALTER ROLE
```

Alice does not get all the rights of the group role automatically, but she can switch to the group role to use them:

```
postgres=# \c - alice
```

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

```
alice=> SET ROLE student;
```

```
SET
```

```
alice=> ALTER ROLE bob LOGIN;
```

```
ALTER ROLE
```

This functions similarly to the su command in Unix.

There are functions that show who is the active session user and what role they are currently switched to:

```
alice=> SELECT session_user, current_user;
```

session_user	current_user
alice	student

(1 row)

Switch back to the old role:

```
alice=> RESET ROLE;
```

```
RESET
```

```
alice=> SELECT session_user, current_user;
```

session_user	current_user
alice	alice

(1 row)

And verify that our commands have been recorded:

```
student$ tail -n 5 /var/log/postgresql/postgresql-13-main.log
```

2024-03-07	13:51:01.904	MSK	[137685]	alice@access_roles	LOG:	duration: 0.446 ms	statement: SET ROLE student;
2024-03-07	13:51:01.969	MSK	[137685]	alice@access_roles	LOG:	duration: 2.674 ms	statement: ALTER ROLE bob LOGIN;
2024-03-07	13:51:02.026	MSK	[137685]	alice@access_roles	LOG:	duration: 0.142 ms	statement: SELECT session_user, current_user;
2024-03-07	13:51:02.083	MSK	[137685]	alice@access_roles	LOG:	duration: 0.058 ms	statement: RESET ROLE;
2024-03-07	13:51:02.110	MSK	[137685]	alice@access_roles	LOG:	duration: 0.095 ms	statement: SELECT session_user, current_user;

Object owner

the role that created the object
(as well as the role's members)

can be changed with the `ALTER . . . OWNER TO role` command

When a role creates any objects in a database, it becomes their *owner*. In addition to that, any members of this role also become owners of these objects.

If necessary, the owner of an object can be changed with the `ALTER` command for the object with the `OWNER TO` clause.

The concept of ownership is especially important when discussing privileges, the next topic of this module.

Owners

When Alice creates a database object, she becomes its owner.

```
alice=> CREATE TABLE test(id integer);
```

CREATE TABLE

An owner of an object is listed under the owner column in the table:

```
alice=> \dt test
```

```
      List of relations
Schema | Name | Type  | Owner
-----+-----+-----+-----
public | test | table | alice
(1 row)
```

Dropping roles

A role can be dropped only if does not own any objects.

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

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

```
student=# DROP ROLE alice;
```

```
ERROR:  role "alice" cannot be dropped because some objects depend on it
DETAIL:  owner of table test
```

To drop the role alice, you must first transfer ownership of her objects to another role:

```
student=# REASSIGN OWNED BY alice TO bob;
```

REASSIGN OWNED

```
student=# \dt test
```

```
      List of relations
Schema | Name | Type  | Owner
-----+-----+-----+-----
public | test | table | bob
(1 row)
```

```
student=# DROP ROLE alice;
```

DROP ROLE

Or you can just drop the owned objects:

```
student=# DROP OWNED BY bob;
```

DROP OWNED

```
student=# DROP ROLE bob;
```

DROP ROLE

Remember that a role may own objects across different databases.

Takeaways



Roles can be considered as both users and groups of users

The properties of a role are defined by its attributes

Roles can be members of other roles

Each database object has an owner role

1. Create a role *swan* without login privileges, but with the rights to create databases and roles.
Create a user *duckling* with login privileges.
2. Verify that *duckling* cannot create a database.
3. Grant *duckling* membership in the *swan* group.
Create a new database as *swan*.

1. Create roles

```
student=# CREATE ROLE swan WITH CREATEDB CREATEROLE;  
CREATE ROLE  
student=# CREATE ROLE duckling WITH LOGIN;  
CREATE ROLE
```

2. Check if the role Duckling can create databases

```
student=# \c - duckling  
You are now connected to database "student" as user "duckling".  
duckling=> CREATE DATABASE access_roles;  
ERROR: permission denied to create database
```

3. Grant membership

```
duckling=> \c - student  
You are now connected to database "student" as user "student".  
student=# GRANT swan TO duckling;  
GRANT ROLE  
student=# \c - duckling  
You are now connected to database "student" as user "duckling".  
duckling=> SET ROLE swan;  
SET  
duckling=> CREATE DATABASE access_roles;  
CREATE DATABASE
```

1. Create roles *alice* and *bob* with login privileges.
Create a table on behalf of *alice*.
2. Set it up so that both roles could modify the table structure
(for example, add columns with the ALTER TABLE command).

2. Only the owners of a table can change its structure. You need to get not only Alice, but also Bob to be an owner of the table.

1. Table and roles

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

```
CREATE ROLE
```

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

```
CREATE ROLE
```

```
student=# \c access_roles alice
```

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

```
alice=> CREATE TABLE test (id integer);
```

```
CREATE TABLE
```

2. Adding a table owner

For Bob to be able to modify the structure of the table, he must become its owner. This can be achieved by including bob into the role of alice. Alice can do it with the following command:

```
alice=> GRANT alice TO bob;
```

```
GRANT ROLE
```

```
alice=> \du alice|bob
```

List of roles		
Role name	Attributes	Member of
alice		{}
bob		{alice}

Now, Bob can add new columns to the table:

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

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

```
bob=> ALTER TABLE test ADD description text;
```

```
ALTER TABLE
```

Or even drop the table:

```
bob=> DROP TABLE test;
```

```
DROP TABLE
```