

Access Control Overview



Copyright

© Postgres Professional, 2017–2026

Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Cover photo by Oleg Bartunov (Nepal peak, 7143 m, Lhonak)

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 and Attributes
Connecting to a Server
Password Authentication
Privileges and Ownership
Privilege Management
Granting Membership
Switching to a Role
Obtaining Privileges

Roles and Attributes



Role is an account in DBMS

can be included into other roles
not associated with the OS user

Attributes define the properties of a role

LOGIN	can log in
SUPERUSER	superuser privileges
CREATEDB	can create databases
CREATEROLE	can create roles
and others	

3

Role in PostgreSQL is an account which is essentially a named set of access rights. Role can be a member of another role — it is convenient when setting up access.

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/18/database-roles>

A role has several *attributes* which define its general properties and rights (unrelated to particular objects).

Generally, attributes come in two opposite variations; for example, CREATEDB (can create databases) and NOCREATEDB (not allowed to create databases).

A role with the LOGIN attribute is considered a user role. A role with NOLOGIN cannot connect to the server and is typically used for grouping other roles.

The table lists only some of the possible attributes.

<https://postgrespro.com/docs/postgresql/18/role-attributes>

<https://postgrespro.com/docs/postgresql/18/sql-createrole>

Roles and Attributes

Create a role for the user Alice, specifying two attributes:

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

```
CREATE ROLE
```

Always pay attention to the role that executes the commands. The name of the current role is in the prompt.

You can get the list of roles with the meta-command:

```
student=# \du
```

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

Note that the student role is a superuser. This is why we did not have any access issues so far.

Create a database for this lesson:

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

```
CREATE DATABASE
```

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

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

Connecting to a Server



1. The lines of `pg_hba.conf` are searched from top to bottom
2. The first line that corresponds to the provided connection settings (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	scram-sha-256
	host	all	all	::1/128	scram-sha-256
...					
	local — socket		all — any role		
	host — TCP/IP		role name		
...					
		all — any DB		all — any IP	
		database name		IP/mask	
				domain name	



5

For each new client, the server has to evaluate whether a database connection should be allowed. Connection settings 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, either via the `pg_reload_conf()` SQL function or the reload command of the management utility.

When a new client appears, the server reads the configuration file from top to bottom to find the line 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) or host (a TCP/IP connection).

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

User: all or the name of a particular role.

Address: all, IP address range, or a domain name. The address is omitted for the local connection type. By default, PostgreSQL listens only for connections coming from 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/18/client-authentication>

Connecting to a Server



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	scram-sha-256
	host	all	all	:::1/128	scram-sha-256

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

6

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

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

Thus, more specific connection lines should precede more generic ones.

There are a lot of different authentication methods:

<https://postgrespro.com/docs/postgresql/18/auth-methods>

Here are 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 connections.

The *scram-sha-256* method asks for a password and checks 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).

Password Authentication



At the server side

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

Providing password on the client

- enter manually
- PGPASSWORD environment variable
- ~/.pgpass file (lines format: *node:port:database:role:password*)

7

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

Password hashes are stored in the pg_authid table in the system catalog.

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

First, the password can be set in the PGPASSWORD environment variable (on the client). 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 (chmod 600), otherwise PostgreSQL will ignore it.

Connection

In order for a role to connect to the database, it must have both the LOGIN attribute and the permission in the file `pg_hba.conf`. You can usually find it right beside the main configuration file:

```
student=# SHOW hba_file;

          hba_file
-----
/etc/postgresql/18/main/pg_hba.conf
(1 row)
```

And you can read it directly from SQL (the contents may vary depending on the server build):

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

type	database	user_name	address	auth_method
local	{all}	{all}		trust
host	{all}	{all}	127.0.0.1	scram-sha-256
host	{all}	{all}	::1	scram-sha-256
local	{replication}	{all}		trust
host	{replication}	{all}	127.0.0.1	scram-sha-256
host	{replication}	{all}	::1	scram-sha-256

(6 rows)

We will use a TCP/IP (host) connection to localhost. Such connection corresponds to the second string in the output, expecting password-based authentication.

The role `alice` was created with a password right away, but the password can be changed at any time:

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

ALTER ROLE

Attempt to connect using a connection string with all the info:

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

```
alice=> \conninfo
```

Connection Information	
Parameter	Value
Database	access_overview
Client User	alice
Host	localhost
Host Address	127.0.0.1
Server Port	5432
Options	
Protocol Version	3.0
Password Used	true
GSSAPI Authenticated	false
Backend PID	15853
SSL Connection	true
SSL Library	OpenSSL
SSL Protocol	TLSv1.3
SSL Key Bits	256
SSL Cipher	TLS_AES_256_GCM_SHA384
SSL Compression	false
ALPN	postgresql
Superuser	off
Hot Standby	off

(19 rows)

Success!

Privilege and Ownership



Privilege

grants the role permission to perform specific actions on an object

Object owner

initially — the role that created it, can be reassigned

receives all privileges on that object

has the right to perform actions not governed by privileges

A privilege on an object grants the role permission to perform specific actions on that object. For example, the `SELECT` privilege on a table allows the role to read data from it.

When a role creates an object, it becomes the owner of that object by default. If needed, the object's owner can be changed using the `ALTER` command with the `OWNER TO` clause.

Some actions are not governed by privileges. These include actions such as dropping or altering an object, granting and revoking privileges, and building an index on a table. Only the owner — besides superusers — can perform these actions, and this right cannot be taken away from them.

In addition to these inherent rights, when an object is created, its owner receives all possible privileges on it. Formally, a superuser can revoke these privileges, but the owner can always restore them, since they have the right to grant privileges — including to themselves.

Privileges



Tables and views

SELECT	read data	} can be set per column
INSERT	insert rows	
UPDATE	change rows	
REFERENCES	foreign key (for tables)	
DELETE	delete rows	
TRUNCATE	empty (for tables)	
TRIGGER	create triggers	
MAINTAIN	perform VACUUM, ANALYZE, REINDEX etc. (for tables)	

10

There are different privileges available for different object types. This slide and the following one list privileges for basic database objects.

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

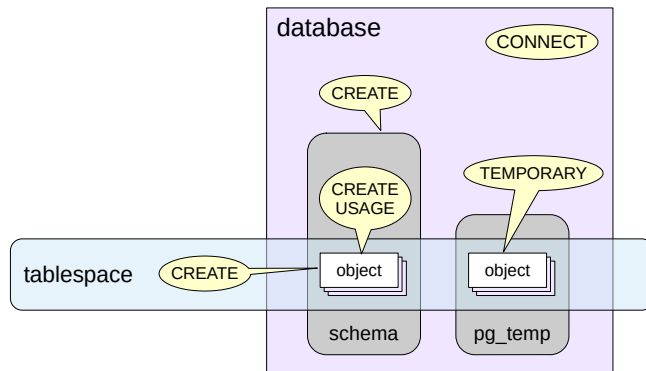
<https://postgrespro.com/docs/postgresql/18/ddl-priv>

<https://postgrespro.com/docs/postgresql/18/sql-grant>

Privileges



Tablespaces, databases, schemas



11

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

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

Since the exact name of the schema for temporary objects is unknown in advance, the privilege to create temporary tables has been moved to the database level (TEMPORARY).

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

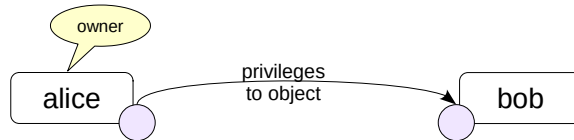
The CONNECT database privilege is required (in addition to pg_hba.conf settings) to be able to connect to this database.

Privilege Management



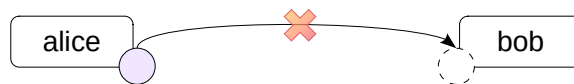
Granting privileges

```
alice=# GRANT privileges TO bob;
```



Revoking privileges

```
alice=# REVOKE privileges FROM bob;
```



12

Privileges are granted using the GRANT command and revoked using the REVOKE command. To grant or revoke a privilege on an object, you must have the right to do so. The owner has this right by default.

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

<https://postgrespro.com/docs/postgresql/18/sql-grant>

<https://postgrespro.com/docs/postgresql/18/sql-revoke>

Privilege Management

Alice has connected to the database. Now she wants to create a schema and some objects.

```
|  alice=> CREATE SCHEMA alice;
|  ERROR:  permission denied for database access_overview
```

What's the issue?

Alice does not have the privilege to create schemas in the database. Grant it:

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

GRANT

Try again:

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

Now, since Alice is the owner of the schema, she has all the privileges on it and can create any objects in it. This schema will be used by default:

```
|  alice=> SELECT current_schemas(false);
|
|  current_schemas
|  -----
|  {alice,public}
|  (1 row)
```

Alice creates 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
```

The superuser creates another role for the user Bob, who will access objects belonging to Alice.

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

CREATE ROLE

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

Bob attempts to access the table t1:

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

What happened?

Bob is neither a superuser nor the owner of this schema, so access is denied.

This meta-command lists current access rights (Access privileges column):

```
|  alice=> \dnx+
```

```

List of schemas
-[ RECORD 1 ]-----+-----
Name           | alice
Owner          | alice
Access privileges |
Description    |
-[ RECORD 2 ]-----+-----
Name           | public
Owner          | pg_database_owner
Access privileges | pg_database_owner=UC/pg_database_owner+
               | =U/pg_database_owner
Description    | standard public schema

```

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

Each privilege is encoded with a single character. In particular, for schemas:

- U = usage;
- C = create.

If the role name is omitted (as in the last line), the record lists privileges for all roles. Note that, except for the database owner (pg_database_owner), every user has only USAGE privilege on the public schema.

If the entire field is omitted (as in the first line), then the default privileges are implied: Alice gets both available privileges on her schema, and the other roles do not get any.

Grant Bob access to the schema. Alice can do that as the owner.

```

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

GRANT

```

Bob attempts to access the table t1 again:

```

bob=> SELECT * FROM alice.t1;

ERROR:  permission denied for table t1

```

Another error. What happened now?

This time, Bob has access to the schema, but not to the table itself. This meta-command shows who has access to the table:

```

alice=> \dp alice.t1

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

```

The field is empty: only the owner, Alice, has access.

Alice grants Bob read and update access:

```

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

GRANT

```

And read and insert rights for one column in the second table:

```

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

GRANT

```

The privileges have changed:

```

alice=> \dp alice.*

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

```

Now the empty field has “appeared” and contains a complete list of privileges. Below are the designations, not all of them are quite obvious:

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

Privileges for columns are displayed separately (under Column privileges).

This time Bob is successful. He adds the schema name to his search path, so that he does not have to type it in every time.

```
|| bob=> ALTER ROLE bob SET search_path = public, alice;
|| ALTER ROLE
```

Now, the search path will be set in each of Bob's sessions.

```
|| bob=> \c
|| You are now connected to database "access_overview" as user "bob".
|| bob=> SHOW search_path;
||
|| search_path
|| -----
|| public, alice
|| (1 row)
||
|| bob=> UPDATE t1 SET n = n + 1;
|| UPDATE 1
|| bob=> SELECT * FROM t1;
||
|| n
|| ---
|| 2
|| (1 row)
```

However, all other operations remain restricted.

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

And the second table:

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

Reading of the other column is not permitted:

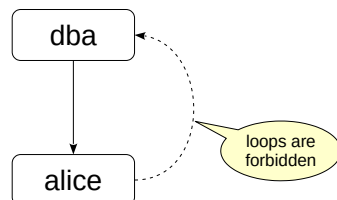
```
|| bob=> SELECT * FROM t2;
|| ERROR: permission denied for table t2
```

Granting Membership



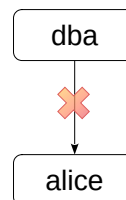
Inclusion to role

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



Exclusion from role

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



The PUBLIC pseudo-role implicitly includes all other roles

14

A role can be granted membership in another role. In this case, the second role acts as a group, and the first role is said to become a member of it. A role can be a member of multiple roles; in turn, a member role may include other roles, but circular dependencies are not permitted.

By default, a role inherits the privileges of any group roles it is a member of. This behavior can be changed by using the NOINHERIT attribute for the role. This makes the user have to explicitly switch to the group role using SET ROLE in order to use its privileges. Role attributes are not inherited, but it is possible to switch to a parent role to use its attributes.

Roles that include other roles typically have the NOLOGIN attribute and are referred to as “group 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. This simplifies access control and administration.

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.

<https://postgrespro.com/docs/postgresql/18/role-membership>

Switching to a Role

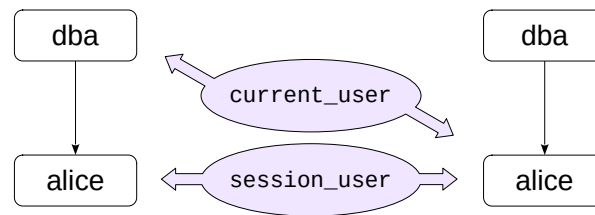


Switching to a role

alice=> SET ROLE dba;

Returning to the original role

alice=> RESET ROLE;



15

By default a role can execute the SET ROLE command to (temporarily) switch to a group role, gaining all of its attributes and object privileges. All actions will be performed on behalf of the group role, and any objects created will be owned by that group role. The privileges of the original role are not available while doing so.

The session_user function returns the name of the user who connected to the server. The name of the current role can be obtained using the current_user function (or current_role, which is the same thing).

<https://postgrespro.com/docs/postgresql/18/functions-info#FUNCTIONS-INFO-ACCESS-TABLE>

Obtaining Privileges



Superuser Access

superusers are not restricted in their actions

Ownership

Holding a Privilege

Role Membership

inheritance of the group role's privileges

switching to the group role

16

A role has several ways to obtain the right for particular actions on an object.

Superusers (roles with the SUPERUSER attribute) are a special case: no privilege checks are performed for them.

By becoming the owner of an object, a role gains the right to perform virtually any action on it.

A role may also gain the right to perform an action on an object by being granted a privilege on that object (either automatically upon creating the object, or explicitly).

Finally, a role can gain additional rights by being made a member of another role, in which case the role inherits group role's privileges by default. The role can switch to the group role, thus gaining not only its privileges but also attributes.

To check if a role has the necessary privilege with respect to some object, you can use the `has_*_privilege` functions:

<https://postgrespro.com/docs/postgresql/18/functions-info>

Group Roles

PostgreSQL provides a number of predefined roles that have access to some frequently used, but privileged capabilities and data. Membership in these roles may be granted to users in order to facilitate administrative and maintenance tasks without providing them superuser capabilities.

This is a list of all roles in the cluster:

```
student=# \duS
```

List of roles	
Role name	Attributes
alice	
bob	
pg_checkpoint	Cannot login
pg_create_subscription	Cannot login
pg_database_owner	Cannot login
pg_execute_server_program	Cannot login
pg_maintain	Cannot login
pg_monitor	Cannot login
pg_read_all_data	Cannot login
pg_read_all_settings	Cannot login
pg_read_all_stats	Cannot login
pg_read_server_files	Cannot login
pg_signal_autovacuum_worker	Cannot login
pg_signal_backend	Cannot login
pg_stat_scan_tables	Cannot login
pg_use_reserved_connections	Cannot login
pg_write_all_data	Cannot login
pg_write_server_files	Cannot login
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS
student	Superuser, Create role, Create DB, Replication, Bypass RLS

The pg_monitor role includes three other predefined roles, it is convenient for setting up monitoring tools. We can see it with \drg meta-command:

```
student=# \drgS
```

List of role grants			
Role name	Member of	Options	Grantor
pg_monitor	pg_read_all_settings	INHERIT, SET	postgres
pg_monitor	pg_read_all_stats	INHERIT, SET	postgres
pg_monitor	pg_stat_scan_tables	INHERIT, SET	postgres

(3 rows)

Bob could not read the second column:

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

The superuser includes Bob into the predefined role role pg_read_all_data:

```
student=# GRANT pg_read_all_data TO bob;
```

GRANT ROLE

Now Bob is a member of the predefined role...

```
student=# \drg
```

List of role grants			
Role name	Member of	Options	Grantor
bob	pg_read_all_data	INHERIT, SET	postgres

(1 row)

...and can read all tables as if he had been granted SELECT privileges for all tables and USAGE privileges for all schemas:

```
|| bob=> SELECT * FROM t2;
```

n	who
1	alice
100	bob
(2 rows)	

The Options column shows attributes for the membership. For Bob as a member of `pg_read_all_data`, SET means the right to switch to the group role, and INHERIT means that Bob can use the group privileges without explicitly switching to the group.

Exclude Bob from the group:

```
student=# REVOKE pg_read_all_data FROM bob;
```

```
REVOKE ROLE
```

Takeaways



Roles, attributes and privileges together form a flexible mechanism that allows to set up access control in different ways

- can grant all access widely

- can restrict access heavily if necessary

When creating a new role, you need to ensure that it can connect to the server

Practice



1. Set up access so that Alice and Bob have their personal schemas in which they can create and use their own tables with no restrictions. Neither Alice nor Bob must have access to each other's objects.
2. Create a table in public schema and let Alice and Bob read its content, but not modify it.
3. Think of another solution to task 2.

19

1. Remember the default setting for *search_path* configuration parameter (see Logical Structure lesson of Data Organization module).
3. Use group role. It looks like an overkill in this situation, but imagine that instead of two users there are hundreds of them, each requiring the same set of dozen privileges.

1. Personal Schemas

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

```
CREATE DATABASE
```

```
student=# CREATE USER alice PASSWORD 'alicepass'; -- same as CREATE ROLE ... LOGIN
```

```
CREATE ROLE
```

```
student=# CREATE USER bob PASSWORD 'bobpass';
```

```
CREATE ROLE
```

The superuser creates schemas for users:

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

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

```
student=# CREATE SCHEMA alice;
```

```
CREATE SCHEMA
```

```
student=# CREATE SCHEMA bob;
```

```
CREATE SCHEMA
```

Users must be owners of their schemas to have no restrictions:

```
student=# ALTER SCHEMA alice OWNER TO alice;
```

```
ALTER SCHEMA
```

```
student=# ALTER SCHEMA bob OWNER TO bob;
```

```
ALTER SCHEMA
```

Now the users can create tables and other database objects, and these objects goes by default to their personal schema:

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

```
|  alice=> CREATE TABLE t(  
|          msg text  
|  );  
|  
|  CREATE TABLE  
|  
|  alice=> INSERT INTO t(msg) VALUES ('I am Alice');  
|  
|  INSERT 0 1
```

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

```
||  bob=> CREATE TABLE t(  
||      msg text  
||  );  
||  
||  CREATE TABLE  
||  
||  bob=> INSERT INTO t(msg) VALUES ('I am Bob');  
||  
||  INSERT 0 1
```

```
student=# \d *.t
```

```
          Table "alice.t"  
Column | Type | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
msg    | text |           |          |
```

```
          Table "bob.t"  
Column | Type | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
msg    | text |           |          |
```

Alice cannot access Bob's table, because she has no access to Bob's schema (and vice versa). Verify:

```
|  alice=> SELECT * FROM bob.t;
```

```
ERROR: permission denied for schema bob
LINE 1: SELECT * FROM bob.t;
          ^
```

2. Common Table

```
student=# CREATE TABLE shared(
        msg text
    );
```

CREATE TABLE

```
student=# INSERT INTO shared(msg) VALUES ('Common table');
```

```
INSERT 0 1
```

Table is created, but the users cannot access it yet:

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

```
ERROR: permission denied for table shared
```

Grant permissions to read the table:

```
student=# GRANT SELECT ON shared TO alice;
```

GRANT

```
student=# GRANT SELECT ON shared TO bob;
```

GRANT

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

```
      msg
-----
Common table
(1 row)
```

```
bob=> SELECT * FROM shared;
```

```
      msg
-----
Common table
(1 row)
```

Neither Alice nor Bob can modify the table:

```
alice=> UPDATE shared SET msg = 'Alice was here';
```

```
ERROR: permission denied for table shared
```

3. Group Role

Another way is to create a group role, which can be handy in case of many users and complex set of permissions.

First, revoke previously granted privileges:

```
student=# REVOKE SELECT ON shared FROM alice;
```

REVOKE

```
student=# REVOKE SELECT ON shared FROM bob;
```

REVOKE

Now, create a role and grant it with necessary privileges:

```
student=# CREATE ROLE read_shared; -- NOLOGIN
```

CREATE ROLE

```
student=# GRANT SELECT ON shared TO read_shared;
```

GRANT

And then grant the new role to the users:

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

GRANT ROLE

```
student=# GRANT read_shared TO bob;
```

GRANT ROLE

Verify:

```
|  alice=> SELECT * FROM shared;

      msg
-----
Common table
(1 row)
```

```
||  bob=> SELECT * FROM shared;

||      msg
||  -----
||  Common table
||  (1 row)
```