

# Access control Privileges



## 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](mailto: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.

- Types of privileges for different objects
- Role categories in terms of access control
- Group privileges
- Granting, revoking and transferring privileges
- Default privileges
- Access control examples

Privileges define the access rights of roles to objects

Tables and views

SELECT	read data	} can be set at the column level
INSERT	insert rows	
UPDATE	change rows	
REFERENCES	foreign key	
DELETE	delete rows	
TRUNCATE	empty a table	
TRIGGER	create triggers	

Privileges define the relationships between cluster objects and roles. They limit the actions the roles can perform on the objects.

The list of possible privileges depends on the object type. Privileges for the main object types are listed on this and the following slide.

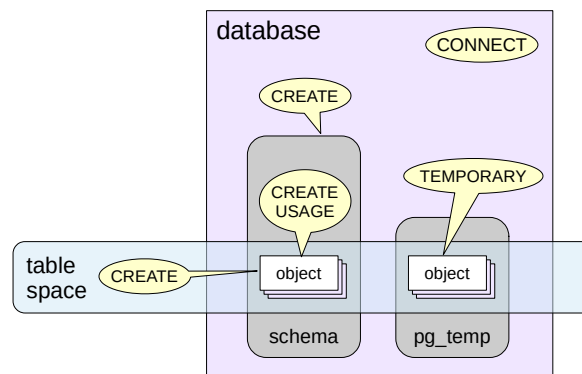
Most privileges are defined for tables and views. Some of them can be defined not only for the entire relation, but also for individual columns.

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

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

# Privileges

Tablespaces,  
databases, schemas



Sequences

SELECT	currval		
UPDATE		nextval	setval
USAGE	currval	nextval	

Sequences have a somewhat unexpected set of privileges. They serve to allow or restrict access to the three control functions.

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 allows connection to this database.

## Superusers

full access to all objects, no checks performed

## Owners

access within privileges

(initially receives all privileges)

actions that are not regulated by privileges, such as deleting objects, granting and revoking privileges, etc.

## Other roles

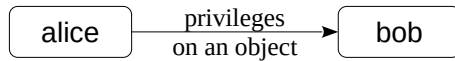
access exclusively within the granted privileges

Generally speaking, a role's ability to access an object is defined by the role's privileges. However, there are three categories of roles that function differently in that regard.

1. Roles with the superuser attribute are the most straightforward: they can do anything and bypass all access control checks.
2. The owner of an object immediately receives a full set of privileges for the object. Technically, these privileges can be revoked, but the owner always retains inherent rights to the object that are not regulated by any privileges. In particular, the owner can grant and revoke privileges (including to and from themselves), delete the object, etc.
3. All other roles have access to the object only as far as the privileges granted to them allow it.

## Granting privileges

```
alice=> GRANT privileges ON object TO bob;
```



the same privilege can be independently granted by multiple roles

## Revoking privileges

```
alice=> REVOKE privileges ON object FROM bob;
```

The owner of an object (and the superuser) has the right to grant and revoke privileges on the object.

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/13/sql-grant>

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

## Privileges

```
student=# CREATE DATABASE access_privileges;
```

```
CREATE DATABASE
```

```
student=# \c access_privileges
```

You are now connected to database "access\_privileges" as user "student".

In this example, Alice will own several objects in her schema.

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

```
CREATE ROLE
```

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

```
CREATE SCHEMA
```

```
student=# GRANT CREATE, USAGE ON SCHEMA alice TO alice;
```

```
GRANT
```

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

You are now connected to database "access\_privileges" as user "alice".

Alice creates two tables.

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

```
CREATE TABLE
```

```
alice=> CREATE TABLE t2(n integer, m integer);
```

```
CREATE TABLE
```

The second role, Bob, will try to access Alice's objects.

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

You are now connected to database "access\_privileges" as user "student".

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

```
CREATE ROLE
```

Bob tries to access t1.

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

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

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

Why does Bob get the error?

Bob has no right to access the table, since he is not a superuser, not the owner of the schema, and does not have the required privileges.

```
student=# \dn+ alice
```

```
          List of schemas  
Name | Owner | Access privileges | Description  
-----+-----+-----+-----  
alice | student | student=UC/student+ |  
      |         | alice=UC/student    |  
(1 row)
```

In each access privileges row is a role, its privileges and the user who granted them:

role=privileges/granter

Privilege names are denoted by single letters. Schema privileges:

- U = usage
- C = create

Alice must grant Bob access to her schema.

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

You are now connected to database "access\_privileges" as user "alice".

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

```
WARNING: no privileges were granted for "alice"
GRANT
```

Why couldn't Alice grant the privilege?

Because Alice does not own the schema.

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

```

          List of schemas
Name | Owner | Access privileges | Description
-----+-----+-----+-----
alice | student | student=UC/student+|
      |         | alice=UC/student   |
(1 row)
```

Make Alice the owner:

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

You are now connected to database "access\_privileges" as user "student".

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

```
ALTER SCHEMA
```

```
student=# \dn+ alice
```

```

          List of schemas
Name | Owner | Access privileges | Description
-----+-----+-----+-----
alice | alice | alice=UC/alice    |
(1 row)
```

Now, Alice can grant access to Bob:

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

You are now connected to database "access\_privileges" as user "alice".

```
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's the problem this time?

Bob has access to the schema, but not to the table.

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

```

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

The empty access privileges field means that the owner has the full set of privileges, and nobody else has any.

Alice must grant Bob the right to read the table:

```
alice=> GRANT SELECT ON t1 TO bob;
```

```
GRANT
```

Let's see how the privileges have changed:

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



Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+ bob=r/alice		

(1 row)

The empty field has filled up, showing that Alice has the full set of privileges. Below are denotations for table privileges (some are not as obvious as the others):

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

Finally, Bob can access the table:

```
bob=> SELECT * FROM alice.t1;
      n
      ---
      (0 rows)
```

But he still cannot add row into it:

```
bob=> INSERT INTO alice.t1 VALUES (42);
ERROR: permission denied for table t1
```

Some privileges can be granted for specific columns:

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

GRANT

```
alice=> GRANT SELECT(m) ON t2 TO bob;
```

GRANT

```
alice=> \dp t2
```

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t2	table		n: bob=a/alice m: bob=ar/alice	+ + + 

(1 row)

Now, Bob can add rows into t2:

```
bob=> INSERT INTO alice.t2(n,m) VALUES (1,2);
INSERT 0 1
```

And can only read one column:

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

bob=> SELECT m FROM alice.t2;
      m
      ---
      2
      (1 row)
```

If necessary, Alice can grant Bob all the privileges, without the need to list them individually.

```
alice=> GRANT ALL ON t1 TO bob;
```

GRANT

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

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+ bob=arwdDxt/alice		

(1 row)

Now, Bob can do anything with the table. For example, delete rows:

```
bob=> DELETE FROM alice.t1;
DELETE 0
```

Maybe, drop the whole table?

```
bob=> DROP TABLE alice.t1;
ERROR:  must be owner of table t1
```

Only the table owner (or the superuser) can delete a table. There is no special privilege to allow this.

A role gets the privileges of the group roles it is a member of  
the INHERIT attribute makes privileges inherited automatically  
with the NOINHERIT attribute, an explicit transition of privileges with the SET ROLE command is required

The pseudo-role `public`  
implicitly includes all other roles

A role can receive privileges to access an object not only directly, but also from group roles in which it is included. In order to simplify administration, you can grant the necessary set of privileges to a group role and then include users into that role, providing them with the entire set of privileges at once. A group role can be viewed as a privilege in itself, and group management is done by the same GRANT and REVOKE commands as privilege management.

A role with the INHERIT attribute (on by default) will automatically have the privileges of all the groups it belongs to. This also applies to the pseudo-role `public`, which implicitly includes all roles.

If a role is created with the NOINHERIT attribute explicitly added, then it will have to use the SET ROLE command to switch to the group role in order to benefit from its privileges. In this case, all actions will be performed on behalf of the group role (for example, the group role will own any created objects).

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

# Default roles

<code>pg_signal_backend</code>	— terminate sessions and cancel queries	
<code>pg_read_all_settings</code>	— read configuration parameters	} <code>pg_monitor</code>
<code>pg_read_all_stats</code>	— access statistics	
<code>pg_stat_scan_tables</code>	— access statistics that block access	
<code>pg_read_server_files</code>	— read files on the server	
<code>pg_write_server_files</code>	— write files on the server	
<code>pg_execute_server_programs</code>	— run programs on the server	

PostgreSQL has a number of default roles that possess special privileges required to perform tasks that usually can only be performed by a superuser. The last three roles were first added in PostgreSQL 11.

A complete list of all roles, including default system roles, can be viewed in psql with the `\duS` command.

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

Similarly, custom group roles can be created, for example, to manage backups.

Group privileges

Alice grants public the privilege to modify t2:

```
alice=> GRANT UPDATE ON t2 TO public;
```

GRANT

```
alice=> \dp t2
```

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t2	table	alice=arwdDxt/alice+=w/alice	n: bob=a/alice m: bob=ar/alice	+ + +
(1 row)					

The empty role (left from the = sign) represents public.

Check if Bob can use the privilege.

```
| bob=> UPDATE alice.t2 SET n = n + 1;
| ERROR: permission denied for table t2
```

Why the error?

Before updating t2, required rows need to be selected, and to do that, the role must have the privilege to read the data (at least the column n from the query). Bob, however, may only read the column m.

```
alice=> GRANT SELECT ON t2 TO bob;
```

GRANT

```
alice=> \dp t2
```

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t2	table	alice=arwdDxt/alice+=w/alice bob=r/alice	n: bob=a/alice m: bob=ar/alice	+ + +
(1 row)					

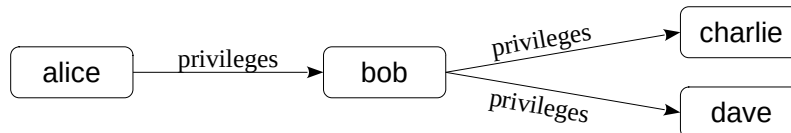
Now, Bob can update the table:

```
| bob=> UPDATE alice.t2 SET n = n + 1;
| UPDATE 1
```

# Right to re-grant

## Granting privileges with the right to re-grant

```
alice=> GRANT privileges ON object TO bob WITH GRANT OPTION;
```



## Revoking privileges

```
alice=> REVOKE privileges ON object FROM bob CASCADE;
```

## Revoking the right to re-grant

```
alice=> REVOKE GRANT OPTION FOR  
privileges ON object FROM bob CASCADE;
```

required  
if the privilege  
was granted  
to other roles

11

When granting certain privileges to a role, you can allow the role to grant (or revoke) these privileges to other roles down the line. This is done with the `GRANT ... WITH GRANT OPTION` command (a similar construction `WITH ADMIN OPTION` for attributes was discussed in the topic “Roles” before). If a role uses this right to re-grant, a hierarchy of roles is formed.

Privileges are revoked with the `REVOKE` command. A role can revoke only those privileges that it has granted to others itself. In the example shown on the slide, `alice` cannot revoke the privilege directly from `charlie` or `dave`.

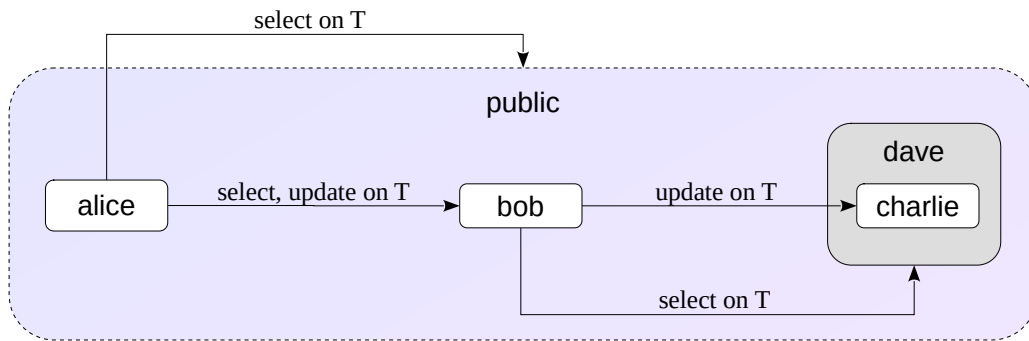
However, if the privilege is revoked from `bob`, it will be automatically revoked from all the roles down the line in the hierarchy. This requires the `CASCADE` keyword (if the hierarchy is not empty, then attempting to revoke without the `CASCADE` keyword will return an error).

The right to re-grant can be revoked without revoking the privilege itself. This is done with the `REVOKE GRANT OPTION FOR` command. The `CASCADE` keyword works here similarly to when revoking a privilege.

## Question

Alice granted the privileges on table T to Bob.

If Alice runs the command  
`REVOKE ALL ON T FROM bob CASCADE`,  
what privileges will Charlie and Dave have?



12

Both Charlie and Dave will end up with the read privilege that they received from public. All privileges granted by Bob to other roles will be revoked.

## Transferring privileges

Create a new role for Charlie and let Bob grant Charlie the privileges for t1, which is owned by Alice.

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

You are now connected to database "access\_privileges" as user "student".

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

CREATE ROLE

Bob has full access to t1:

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

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+		
			bob=arwdDxt/alice		
(1 row)					

But he cannot transfer the privileges to Charlie:

```
| bob=> GRANT SELECT ON alice.t1 TO charlie;
```

```
| WARNING: no privileges were granted for "t1"  
| GRANT
```

Alice must give Bob permission to do that.

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

You are now connected to database "access\_privileges" as user "alice".

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

GRANT

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

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+		
			bob=ar*w*dDxt/alice		
(1 row)					

The asterisks to the right of each privilege character show the right to re-grant them.

Now, Bob can grant the privileges to Charlie, including the right to re-grant them:

```
| bob=> GRANT SELECT ON alice.t1 TO charlie WITH GRANT OPTION;
```

```
| GRANT
```

```
| bob=> GRANT UPDATE ON alice.t1 TO charlie;
```

```
| GRANT
```

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

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+		
			bob=ar*w*dDxt/alice+		
			charlie=r*w/bob		
(1 row)					

A role may be granted the same privilege by multiple other roles. Note that when a privilege is granted by a superuser, it is granted on behalf of the object's owner:

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

You are now connected to database "access\_privileges" as user "student".



```
student=# GRANT UPDATE ON alice.t1 TO charlie;
```

```
GRANT
```

```
student=# \dp alice.t1
```

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+		
			bob=ar*w*dDxt/alice+		
			charlie=r*w/bob	+	
			charlie=w/alice		
(1 row)					

A role can revoke privileges only from those roles it itself granted them to. For example, Alice cannot revoke the privilege to re-grant from Charlie, because it wasn't Alice who granted that privilege to Charlie in the first place.

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

You are now connected to database "access\_privileges" as user "alice".

```
alice=> REVOKE GRANT OPTION FOR SELECT ON alice.t1 FROM charlie;
```

```
REVOKE
```

No error is returned, but no privileges are revoked either:

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

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+		
			bob=ar*w*dDxt/alice+		
			charlie=r*w/bob	+	
			charlie=w/alice		
(1 row)					

At the same time, Alice cannot just revoke privileges from Bob if Bob has granted them to someone else:

```
alice=> REVOKE GRANT OPTION FOR SELECT ON alice.t1 FROM bob;
```

```
ERROR: dependent privileges exist
HINT: Use CASCADE to revoke them too.
```

In this situation, Alice must revoke them hierarchically. This is done with the CASCADE keyword:

```
alice=> REVOKE GRANT OPTION FOR SELECT ON alice.t1 FROM bob CASCADE;
```

```
REVOKE
```

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

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+		
			bob=arw*dDxt/alice +		
			charlie=w/bob +		
			charlie=w/alice		
(1 row)					

Here, Bob lost his privilege to re-grant, and Charlie lost the privilege itself.

Privileges can be revoked hierarchically in a similar manner.

```
alice=> REVOKE SELECT ON alice.t1 FROM bob CASCADE;
```

```
REVOKE
```

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

Access privileges					
Schema	Name	Type	Access privileges	Column privileges	Policies
alice	t1	table	alice=arwdDxt/alice+		
			bob=aw*dDxt/alice	+	
			charlie=w/bob		
			charlie=w/alice		
(1 row)					



The only privilege for functions and procedures

EXECUTE	execution
---------	-----------

Security features

SECURITY INVOKER	executed with the calling role's rights (by default)
------------------	---

SECURITY DEFINER	executed with the owner's rights
------------------	----------------------------------

The privilege EXECUTE allows users to execute routines (functions and procedures).

The user on behalf of which the routine is executed is important. If a routine is declared as a SECURITY INVOKER (by default), it is executed with the rights of the user that executes it. In this case, the operators inside the routine can access only those objects that the user has the rights to access.

On the other hand, if declared with the SECURITY DEFINER keyword, the routine will use the rights of its owner. This is a way to allow certain users perform certain actions on objects they personally have no access to.

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

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

By default, the public role gets a number of privileges

for databases	CONNECT (connect to databases) TEMPORARY (create temporary tables)
for the public schema	CREATE (create objects) USAGE (access objects)
for pg_catalog and information_schema	USAGE (access objects)
for routines	EXECUTE (run routines)

Convenient, but not really secure

15

By default, the pseudo-role `public` has a number of privileges (this means that all roles get them):

- connecting and creating temporary tables for **all** databases,
- using the **public** schema and creating objects in it,
- using schemas **pg\_catalog** and **information\_schema**,
- executing **all** functions and procedures.

(In PostgreSQL 15, the public role loses the right to create objects.)

Such behavior may be undesirable. In this case, you must explicitly revoke some of the privileges from `public`. You can also do so in the template database `template1`, so that the changes persist in any newly created databases. However, revoking routine rights demands the use of the default privileges mechanism, which will be discussed below.

## Subroutines

Alice creates a simple function that returns the number of rows in the table t1:

```
alice=> CREATE FUNCTION foo() RETURNS bigint AS $$  
  SELECT count(*) FROM t1;  
$$ LANGUAGE sql;
```

CREATE FUNCTION

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

INSERT 0 1

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

```
foo  
-----  
1  
(1 row)
```

The public pseudo-role is automatically granted the EXECUTE privilege for any created function. This is why, for example, Bob may immediately execute the function created by Alice.

This is partially kept in check by the default (or explicitly added) keyword SECURITY INVOKER that makes the function execute with the rights of the user who executes it:

```
| bob=> SET search_path = public, alice;  
| SET  
| bob=> SELECT foo();  
| ERROR: permission denied for table t1  
| CONTEXT: SQL function "foo" statement 1
```

Therefore, Bob cannot access any objects he does not have the privileges to access.

---

Since the function does not explicitly define the schema for the table t1, Bob can create his own t1, and the function will work with the table first found in the search path:

```
| bob=> CREATE TABLE t1(n numeric);
```

```
| CREATE TABLE
```

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

```
| foo  
|-----  
| 0  
| (1 row)
```

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

```
foo  
-----  
1  
(1 row)
```

---

A function can also be made to run with the rights of its creator (SECURITY DEFINER):

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

ALTER FUNCTION

In this case, the function will always run on behalf of the role that has created it. So, Bob can drop his table:

```
| bob=> DROP TABLE t1;
```

```
| DROP TABLE
```

And now have access to Alice's:

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

```
foo
----
1
(1 row)
```

This is where you really have to keep an eye on what privileges are granted to whom. A good idea is to revoke the EXECUTE privilege from public and grant it explicitly to roles that need it.

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

```
REVOKE
```

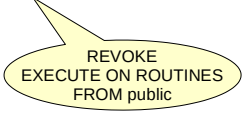
```
bob=> SELECT foo();
```

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

A way to grant or revoke privileges when creating an object

```
ALTER DEFAULT PRIVILEGES
[ FOR ROLE target_roles_list ]
[ IN SCHEMA schema ]
  GRANT privileges ON object_class TO role;

ALTER DEFAULT PRIVILEGES
  REVOKE privileges ON object_class FROM role;
```



REVOKE  
EXECUTE ON ROUTINES  
FROM public

You can define additional privileges to be granted or revoked when creating an object. This is done with the `ALTER DEFAULT PRIVILEGES` command.

The default privileges mechanism triggers when a *target\_role* (the current user by default) creates an object that belongs to the specified *object\_class* (i. e. a table or a function) in the specified *schema* (or in any schema by default). The `GRANT` clause here means that the created object should be granted the specified privileges for the specified role. The `REVOKE` clause, on the other hand, is used to revoke the privileges.

The default privileges mechanism grants `public` the privilege to execute routines when they are created. To avoid this behavior, run the command

```
ALTER DEFAULT PRIVILEGES
FOR ROLE ...
REVOKE EXECUTE ON ROUTINES FROM public;
```

Note that under the `FOR ROLE` clause, all the roles that can create routines must be listed.

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

## Default privileges

To avoid having the public pseudo-role acquiring the right to execute any newly created function, we need to revoke this privilege from it.

Check the current configuration:

```
alice=> \ddp

      Default access privileges
 Owner | Schema | Type | Access privileges
-----+-----+-----+-----
(0 rows)
```

The table looks empty, but we already know that empty fields represent the default values. To see the values, check the documentation or use the following command:

```
alice=> SELECT acldefault('function', 'alice'::regrole);

      acldefault
-----
 {=X/alice,alice=X/alice}
(1 row)
```

This is the value that allows public to execute functions. Revoke it:

```
alice=> ALTER DEFAULT PRIVILEGES
REVOKE EXECUTE ON ROUTINES FROM public;
```

```
ALTER DEFAULT PRIVILEGES
```

Now the field is no longer empty, and the execute privilege for public is gone:

```
alice=> \ddp

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

Now, when Alice creates new functions, Bob will no longer be able to execute them:

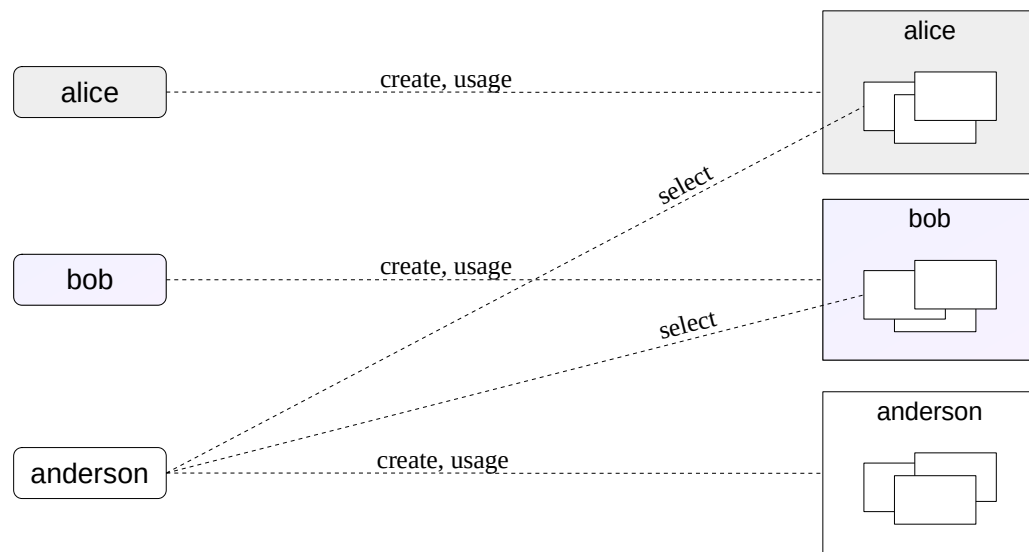
```
alice=> CREATE FUNCTION bar() RETURNS integer AS $$
SELECT 1;
$$ LANGUAGE sql SECURITY DEFINER;
```

```
CREATE FUNCTION
```

```
| bob=> SELECT bar();
| ERROR: permission denied for function bar
```



# Example 1



19

Access control mechanisms (roles, attributes and privileges, and schemas) are flexible enough to let you organize any sort of operations conveniently.

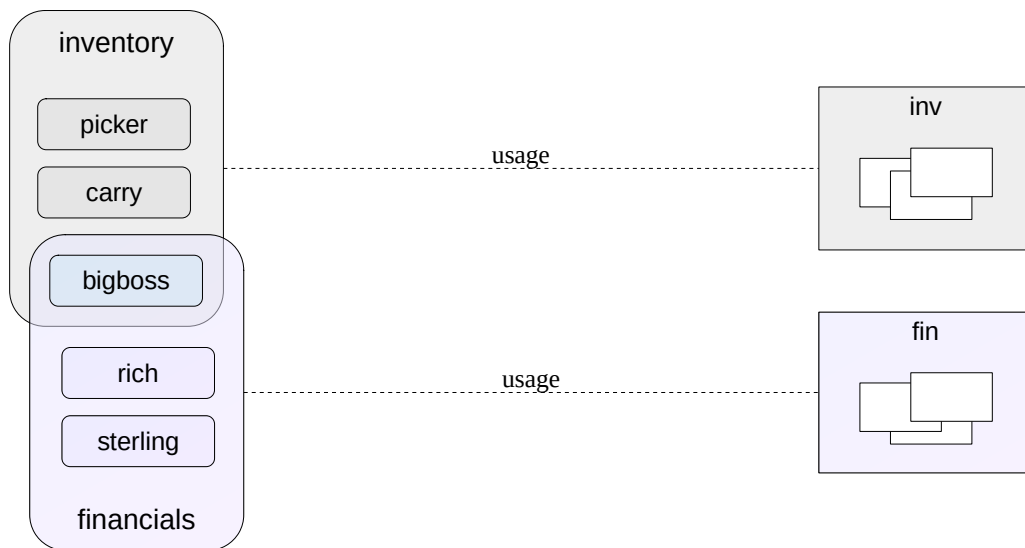
Here's a simple example.

Professor Anderson and his students Alice and Bob are engaged in research. They store their research data in a database.

On the university database server, the administrator created a user and a schema under the same name for each of them. The default research path was never modified. No group roles are used.

Each user owns the objects they create in their schemas. In addition to that, Alice and Bob grant access to some of their tables to Professor Anderson so that he can review their data.

## Example 2



20

A more complex example.

A dedicated DB server runs an ERP system. It comprises an inventory module and a financial module.

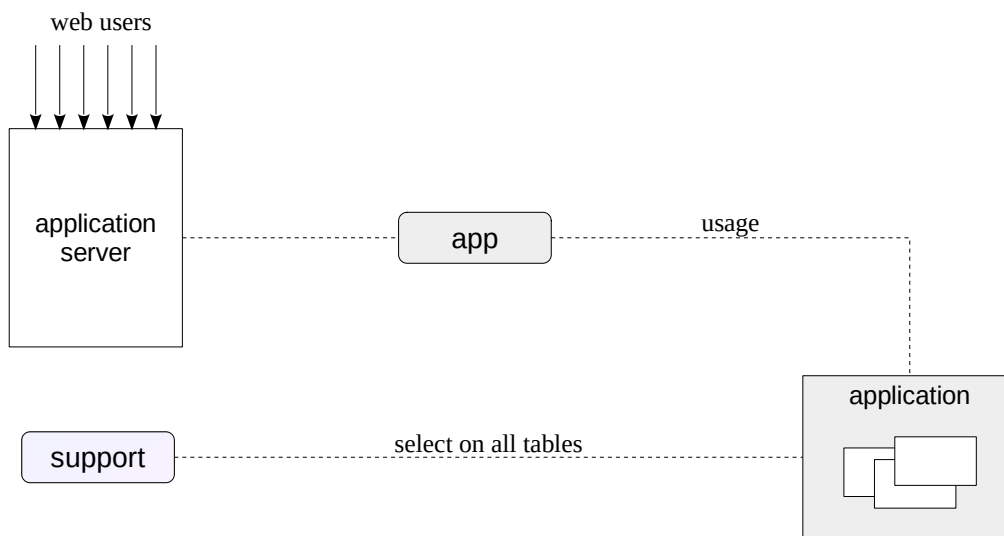
Each module has a schema (`inv` and `fin`) and a group role (`inventory` and `financials`). The group roles own all objects in their corresponding schemas.

The warehouse employs Mr Picker and Mr Putter. Each have a user included into `inventory`.

The financial department employs Mrs Rich and Mr Sterling. Each have a user included into `financials`.

There is also a user for the CEO included into both roles on the off chance that he'd try and use the system.

## Example 3



21

User authentication is often set up on an application server outside of the actual database. With thousands of users and online sign-up functionality, it is more efficient to offload user management from the database to an external service.

In this case, the application server connects to the database under one pre-configured role, and the information about the user is translated as context, if necessary.

But even in this case, the database still needs some roles: A tech support role, for example, that will need the rights to read main server tables to troubleshoot possible issues.

Privileges define the access rights of roles to objects

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

- easy to allow access to everything for everyone

- can restrict access heavily, if necessary

Set up privileges so that some users have full access to the tables, while others can only query, but not modify the data.

1. Create a new database and two roles: `writer` and `reader`.
2. Revoke all privileges for the schema `public` from the role `public`, grant both privileges to `writer`, and only the usage privilege to `reader`.
3. Set up the default privileges so that `reader` gets read access to the tables owned by `writer` in the schema `public`.
4. Create users `w1` in the `writer` group and `r1` in the `reader` group.
5. As `writer`, create a table.
6. Verify that `r1` has read-only access to the table, and `w1` has full access to it, including the ability to remove it.

## 1. Database and roles

```
=> CREATE DATABASE access_privileges;
```

CREATE DATABASE

```
=> CREATE USER writer;
```

CREATE ROLE

```
=> CREATE USER reader;
```

CREATE ROLE

## 2. Privileges

```
=> \c access_privileges
```

You are now connected to database "access\_privileges" as user "student".

```
=> REVOKE ALL ON SCHEMA public FROM public;
```

REVOKE

```
=> GRANT ALL ON SCHEMA public TO writer;
```

GRANT

```
=> GRANT USAGE ON SCHEMA public TO reader;
```

GRANT

## 3. Default privileges

```
=> ALTER DEFAULT PRIVILEGES
FOR ROLE writer
IN SCHEMA public
GRANT SELECT ON TABLES TO reader;
```

ALTER DEFAULT PRIVILEGES

## 4. Users

Writer role:

```
=> CREATE ROLE w1 LOGIN IN ROLE writer;
```

CREATE ROLE

The IN ROLE keyword immediately adds the new role into the specified one. It is equivalent to:

```
CREATE ROLE w1 LOGIN;
```

```
GRANT writer TO w1;
```

Reader role:

```
=> CREATE ROLE r1 LOGIN IN ROLE reader;
```

CREATE ROLE

## 5. Table

```
=> \c - writer
```

You are now connected to database "access\_privileges" as user "writer".

```
=> CREATE TABLE t(n integer);
```

CREATE TABLE

## 6. Verification

w1 can write:

```
=> \c - w1
```

You are now connected to database "access\_privileges" as user "w1".

```
=> INSERT INTO t VALUES (42);
```

```
INSERT 0 1
```

r1 can read the table:

```
=> \c - r1
```

You are now connected to database "access\_privileges" as user "r1".

```
=> SELECT * FROM t;
```

```
 n
----
 42
(1 row)
```

But cannot modify:

```
=> UPDATE t SET n = n + 1;
```

ERROR: permission denied for table t

w1 can drop the table:

```
=> \c - w1
```

You are now connected to database "access\_privileges" as user "w1".

```
=> DROP TABLE t;
```

```
DROP TABLE
```

PostgreSQL 14 adds a pre-configured role `pg_read_all_data` that automatically has read access to all data.

1. Create the role `alice`. Create a table.  
Grant `alice` the privilege to read the table and the privilege to change it with the right to re-grant.
2. View the access rights to the created table using the `table_privileges` view in the information schema.  
Compare with the `\dp` command output.
3. View the access rights to the created table using the `has_table_privileges` function.

2. Other views of the information schema:

<https://postgrespro.com/docs/postgresql/13/information-schema>

3. Other functions for checking privileges:

<https://postgrespro.com/docs/postgresql/13/functions-info#FUNCTIONS-INFO-ACCESS-TABLE>



## 1. Role, table, privileges

```
=> CREATE DATABASE access_privileges;
```

```
CREATE DATABASE
```

```
=> CREATE USER alice;
```

```
CREATE ROLE
```

```
=> \c access_privileges
```

You are now connected to database "access\_privileges" as user "student".

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

```
CREATE TABLE
```

```
=> GRANT SELECT ON test TO alice;
```

```
GRANT
```

```
=> GRANT UPDATE ON test TO alice WITH GRANT OPTION;
```

```
GRANT
```

## 2. Privileges in the information schema

```
=> SELECT grantee, grantor, privilege_type, is_grantable
FROM information_schema.table_privileges
WHERE table_name = 'test';
```

grantee	grantor	privilege_type	is_grantable
student	student	INSERT	YES
student	student	SELECT	YES
student	student	UPDATE	YES
student	student	DELETE	YES
student	student	TRUNCATE	YES
student	student	REFERENCES	YES
student	student	TRIGGER	YES
alice	student	SELECT	NO
alice	student	UPDATE	YES

(9 rows)

The psql command displays the same information differently:

```
=> \dp test
```

Schema	Name	Type	Access privileges	Column privileges	Policies
public	test	table	student=arwdDxt/student+  alice=rw*/student		

(1 row)

## 3. Functions to check privileges

```
=> SELECT has_table_privilege('alice', 'test', 'SELECT') AS has_select,
       has_table_privilege('alice', 'test', 'UPDATE') AS has_update,
       has_table_privilege('alice', 'test', 'DELETE') AS has_delete;
```

has_select	has_update	has_delete
t	t	f

(1 row)