

# Lab VI

## PostgreSQL -User, Role, Authorization, View

23/02/2025

---

### Views (Virtual Tables) in SQL

The relations declared through CREATE TABLE statements are called base tables (or base relations); this means that the relation and its tuples are actually created and stored as a file by the DBMS. Base relations are distinguished from virtual relations, created through the CREATE VIEW statement, which may or may not correspond to an actual physical file.

We can think of a view as a way of specifying a table that we need to reference frequently, even though it may not exist physically.

Consider the company database schema

#### EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----

#### DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------

#### DEPT\_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

#### PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
-------	----------------	-----------	------

#### WORKS\_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

#### DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
-------------	-----------------------	-----	-------	--------------

Suppose we may have to frequently issue queries that retrieve the employee name and the project names that the employee works on. Rather than having to specify the join of the three tables EMPLOYEE, WORKS\_ON, and PROJECT every time we issue this query, we can define a view that is specified as the result of these joins. Then we can issue queries on the view, which are specified as single table retrievals rather than as retrievals involving two joins on three tables. We call the EMPLOYEE, WORKS\_ON, and PROJECT tables the defining tables of the view.

## 1. Creating PostgreSQL Views

To create a view, we use CREATE VIEW statement. The simplest syntax of the CREATE VIEW statement is as follows:

```
CREATE VIEW view_name AS query;
```

Example:

```
CREATE VIEW WORKS_ON_INFO AS (  
SELECT e.first_name||' '||e.last_name EMP_NAME,  
       p.project_name PROJ_NO,  
       w.hours HOURS  
FROM   project p,  
       works_on w,  
       employee e  
WHERE  p.project_no=w.project_no  
       AND w.emp_ssn=e.ssn);
```

Query the view to retrieve info:

Retrieve the name of employee working for project name 'ProductY'.

```
select * from works_on_info where PROJ_NO='ProductY';
```

	emp_name text	proj_no character varying (25)	hours integer
1	John Smith	ProductY	8
2	Joyce English	ProductY	20
3	Franklin Wong	ProductY	10

## 2. Changing PostgreSQL Views

To change the defining query of a view, you use the CREATE VIEW statement with OR REPLACE addition as follows:

```
CREATE OR REPLACE VIEW view_name AS query;
```

Example: add an department number (d\_no) to the works\_on\_info view

```
CREATE OR REPLACE VIEW WORKS_ON_INFO AS (  
SELECT e.first_name||' '||e.last_name EMP_NAME,  
        p.project_name PROJ_NO,  
        w.hours HOURS,p.dept_no DEPT  
  
FROM project p,  
        works_on w,  
        employee e  
WHERE p.project_no=w.project_no  
        AND w.emp_ssn=e.ssn);
```

Query

```
select * from works_on_info where PROJ_NO='ProductY';
```

	emp_name text	proj_no character varying (25)	hours integer	dept integer
1	John Smith	ProductY	8	5
2	Joyce English	ProductY	20	5
3	Franklin Wong	ProductY	10	5

To change the definition of a view, you use the ALTER VIEW statement. For example, you can change the name of the view from works\_on\_info to project\_allocation by using the following statement:

```
ALTER VIEW works_on_info RENAME TO project_allocation_info;
```

Query

```
select * from project_allocation_info where PROJ_NO='ProductY';
```

	emp_name text	proj_no character varying (25)	hours integer	dept integer
1	John Smith	ProductY	8	5
2	Joyce English	ProductY	20	5
3	Franklin Wong	ProductY	10	5

### 3. Removing PostgreSQL Views

To remove an existing view in PostgreSQL, you use DROP VIEW statement as follows:

```
DROP VIEW [ IF EXISTS ] view_name;
```

Removing a view that does not exist in the database will result in an error. To avoid this, you normally add IF EXISTS option to the statement to instruct PostgreSQL to remove the view if it exists, otherwise, do nothing.

For example, to remove the project\_allocation\_info view that you have created, you execute the following query:

```
DROP VIEW IF EXISTS project_allocation_info;
```

The view project\_allocation\_info is removed from the database.

4.Using PostgreSQL DROP VIEW statement to drop a view that has dependent objects

Example:

Consider the views depn\_info, it has info about an employee and his dependents

```
CREATE VIEW depn_info AS(  
  
SELECT e.first_name||' '||e.last_name emp_name,  
       d.first_name dep_name,  
       EXTRACT(YEAR FROM age(d.birth_date)) dep_age,  
       d.relationship relation  
FROM employee e  
JOIN dependent d  
ON e.ssn = d.emp_ssn);
```

Lets create another view to output the details of dependents of a employee whose age is between 18 and 60.

```
CREATE VIEW major_dependent AS(  
  
SELECT emp_name, dep_name, dep_age, relation  
FROM depn_info  
where dep_age > 18 and dep_age < 60 );
```

Lets use the DROP VIEW statement to drop the depn\_info view:

```
DROP VIEW IF EXISTS DEPN_INFO ;
```

```
ERROR: cannot drop view depn_info because other objects depend on it  
DETAIL: view major_dependent depends on view depn_info  
HINT: Use DROP ... CASCADE to drop the dependent objects too.  
SQL state: 2BP01
```

To drop the view depn\_info, you need to drop its dependent object first or use the CASCADE option like this:

```
DROP VIEW IF EXISTS DEPN_INFO  
CASCADE;
```

This statement drops the depn\_info view as well as its dependent object which is the major\_dependent. It issued the following notice:

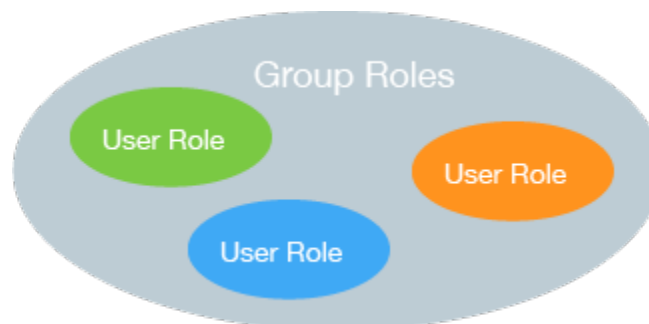
```
NOTICE: drop cascades to view major_dependent
DROP VIEW
```

```
Query returned successfully in 39 msec.
```

A view is supposed to be always up-to-date; if we modify the tuples in the base tables on which the view is defined, the view must automatically reflect these changes. Hence, the view is not realized or materialized at the time of view definition but rather at the time when we specify a query on the view.

## USER ROLE

PostgreSQL uses roles to represent user accounts. Typically, roles that can log in are called login roles. They are equivalent to users in other database systems. When roles contain other roles, they are called group roles.



*/\* PostgreSQL combined the users and groups into roles since version 8.1\*/*

Logging in to a database as a user.

psql -U username -d database      (here username is the name of the role)

```
psql -U postgres dvdrental
```

## PostgreSQL CREATE ROLE statement

To create a new role, you use the CREATE ROLE statement as follows:

```
CREATE ROLE role_name;
```

example:


```
create role bob;
```

```
CREATE ROLE
```

```
Query returned successfully in 35 msec.
```

To get all roles in the current PostgreSQL database server, you can query them from the pg\_roles system catalog as follows:

```
SELECT rolname FROM pg_roles;
```

	rolname name 
1	pg_database_owner
2	pg_read_all_data
3	pg_write_all_data
4	pg_monitor
5	pg_read_all_settings
6	pg_read_all_stats
7	pg_stat_scan_tables
8	pg_read_server_files
9	pg_write_server_files
10	pg_execute_server_program
11	pg_signal_backend
12	pg_checkpoint
13	postgres
14	bob

- The roles that start with pg\_ are system roles.

If you use the **psql** tool, you can use the **\du** command to list all existing roles in the current PostgreSQL database server.

```
postgres=# \du
```

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

As you can see clearly from the output, the role bob cannot login.

To allow the role bob to log in to the PostgreSQL database server, you need to add the LOGIN attribute to it.

## Role attributes



The attributes of a role define privileges for that role including login, superuser, database creation, role creation, password, etc:

```
CREATE ROLE name with options;
```

In this syntax, the WITH keyword is optional. And the option can be one or more attributes including SUPER, CREATEDB, CREATEROLE, etc.

### 1) Create login roles

The following statement creates a role called alice that has the login privilege and an initial password:

```
CREATE ROLE Alice
LOGIN
PASSWORD 'Password1';
```

```
CREATE ROLE
```

```
Query returned successfully in 58 msec.
```

- Password must be placed in single quotes ( ' ).

```
postgres=# \du
```

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

- Now, you can use the role alice to log in to the PostgreSQL database server using the **psql** client tool:

```
psql -U alice -W postgres
```

If you get the error **FATAL: Peer authentication failed for user "alice"**,

Then do the following:

1. Make sure your are out of psql client
2. On the terminal using gedit open the file pg\_hba.conf :

```
sudo gedit /etc/postgresql/15/main/pg_hba.conf
```

3. In the file make the following changes.

```
# Database administrative login by Unix domain socket
local all postgres trust Change peer to trust
# TYPE DATABASE USER ADDRESS METHOD
# "local" is for Unix domain socket connections only
local all all md5 Change peer to md5
# IPv4 local connections:
host all all 127.0.0.1/32 scram-sha-256
# IPv6 local connections:
host all all ::1/128 scram-sha-256
# Allow replication connections from localhost, by a user with the
# replication privilege.
local replication all peer
```

4. Restart the postgresql server: `sudo systemctl restart postgresql`
5. Try to connect again: `psql -U alice -W postgres`

## 2) Create superuser roles

The following statement creates a role called john that has the superuser attribute.

```
CREATE ROLE john
SUPERUSER
LOGIN
PASSWORD 'Password2';
```

- The superuser can override all access restrictions within the database therefore you should create this role only when needed.
- One must be a superuser in order to create another superuser role.

## 3) Create roles that can create databases

If you want to create roles that have the database creation privilege, you use the `CREATEDB` attribute:

```
CREATE ROLE dba  
CREATEDB  
LOGIN  
PASSWORD 'Password3';
```

#### 4) Create roles with validity period

To set a date and time after which the role's password is no longer valid, you use the `valid until` attribute:

```
VALID UNTIL 'timestamp'
```

For example, the following statement creates a `dev_api` role with password valid until the 27/02/2023:

```
CREATE ROLE dev_api WITH  
LOGIN  
PASSWORD 'Password4'  
VALID UNTIL '2023-02-27';
```

#### 5) Create roles with connection limit

To specify the number of concurrent connections a role can make, you use the `CONNECTION LIMIT` attribute:

```
CONNECTION LIMIT connection_count
```

The following creates a new role called `api` that can make 1000 concurrent connections:

```
CREATE ROLE api
LOGIN
PASSWORD 'Password4'
CONNECTION LIMIT 1000;
```

The following **psql** command shows all the roles that we have created so far:

```
postgres=# \du
```

Role name	List of roles Attributes	Member of
alice		{}
api	1000 connections	{}
bob	Cannot login	{}
dba	Create DB	{}
dev_api	Password valid until 2023-02-27 00:00:00+05:30	{}
john	Superuser	{}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}

## PostgreSQL GRANT statement

After creating a role with the LOGIN attribute, the role can log in to the PostgreSQL database server. However, it cannot do anything to the database objects like tables, views, functions, etc.

For example, the user role cannot select data from a table or execute a specific function.

To allow the user role to interact with database objects, you need to grant privileges on the database objects to the user role by using the GRANT statement.

The following shows the simple form of the GRANT statement that grants one or more privileges on a table to a role:

```
GRANT privilege_list | ALL
ON table_name
TO role_name;
```

In this syntax:

1. Privilege\_list can be SELECT, INSERT, UPDATE, DELETE, TRUNCATE, etc.  
You use the ALL option to grant all privileges on a table to the role.
2. Specify the name of the table after the ON keyword.
3. Specify the name of the role to which you want to grant privileges.

Example:

Lets use the role alice created in the previous section to log in to the PostgreSQL database server in a separate session.

```
indu@indu-VirtualBox:~$ psql -U alice postgres
Password for user alice:
psql (15.2 (Ubuntu 15.2-1.pgdg20.04+1))
Type "help" for help.

postgres=> █
```

You can confirm the user by the query 'select current\_user; '

```
week5=> select current_user;
current_user
-----
alice
(1 row)
```

```

week5=> \dt
          List of relations
 Schema |      Name      | Type  | Owner
-----+-----+-----+-----
 public | accounts       | table | postgres
 public | attend         | table | postgres
 public | candidates     | table | postgres
 public | document       | table | postgres
 public | product        | table | postgres
 public | students       | table | postgres
(6 rows)

```

Lets try to retrieve all the tuples in the relation accounts

```

week5=> select * from accounts;
ERROR:  permission denied for table accounts

```

### 1) Grant SELECT privilege on a table to a role

To grant the SELECT privilege on the candidates table to the role joe, you execute the following GRANT statement in the postgres' session:

```

GRANT select
ON accounts
TO alice;

```

Now let's try to retrieve all the tuples in the relation accounts in the alice session:

```

 id | name  | balance
---+-----+-----
  4 | Alice | 10000.00
  3 | bob   | 10000.00
(2 rows)

```

Now let's try to insert a tuple into the relation accounts in the alice session:

```
week5=> insert into accounts(name,balance) values ('Joe',20000);
ERROR:  permission denied for table accounts
```

## 2) Grant a list of privileges on a table to a role

Grant INSERT, UPDATE, and DELETE privileges on the candidates table to the role alice.

```
GRANT insert, delete, update
ON  accounts
TO  alice;
```

Now let's try to insert a tuple into the relation accounts in the alice session:

```
week5=> insert into accounts(name,balance) values ('Joe',20000);
ERROR:  permission denied for sequence accounts_id_seq
```

```
grant usage, select
on sequence accounts_id_seq
to alice;
```

```
week5=> insert into accounts(name,balance) values ('Joe',20000);
INSERT 0 1
week5=> select * from accounts;
 id | name  | balance
----+-----+-----
  3 | bob   |  9500.00
  4 | Alice | 10500.00
  5 | Joe   | 20000.00
(3 rows)
```

## 3) Grant all privileges on a table to a role

The following statement grants all privileges on the candidates table to the role alice:

```
grant all
on accounts
to alice;
```

#### 4) Grant all privileges on all tables in a schema to a role

The following statement grants all privileges on all tables in the public schema of the week5 database to the role alice

```
grant all
on all tables
in schema "public"
to alice;
```

#### 5) Grant SELECT on all tables

Sometimes, you want to create a read-only role that can only select data from all tables in a specified schema. In order to do that, you can grant SELECT privilege on all tables in the public schema like this:

```
grant select
on all tables
in schema "public"
to alice;
```

## PostgreSQL REVOKE statement

The REVOKE statement revokes previously granted privileges on database objects from a role.



```
REVOKE privilege | ALL
ON TABLE table_name | ALL TABLES IN SCHEMA schema_name
FROM role_name;
```

In this syntax:

1. Specify one or more privileges that you want to revoke. You use the ALL option to revoke all privileges.
2. Specify the name of the table after the ON keyword. You use the ALL TABLES to revoke specified privileges from all tables in a schema.
3. Specify the name of the role from which you want to revoke privileges.

Example:

We are already logged in to the PostgreSQL database week5 in a separate session using the role alice which has been granted all privileges.

Evidence:

```
week5=> select * from accounts;
 id | name  | balance
----+-----+-----
  4 | Alice | 10500.00
  3 | bob   |  9000.00
  5 | Joe   | 20500.00
  6 | kyle  | 10000.00
(4 rows)
```

Now, let us revoke the select privilege from alice, in the postgres session .

```
REVOKE select
on accounts
from alice;
```

Lets try to select all tuples from the table accounts.

```
week5=> select * from accounts;
ERROR:  permission denied for table accounts
week5=>
```

REVOKE statement with the ALL option

```
REVOKE ALL
ON accounts
FROM alice;
```

All the privileges are revoked from alice.

```
week5=> update accounts set balance = 20000 where id=6;
ERROR:  permission denied for table accounts
```

## PostgreSQL Role Membership

It is easier to manage roles as a group so that you can grant or revoke privileges from a group as a whole instead of doing it on an individual role.

1. Create a role that represents a group
2. Grant membership in the group role to individual roles.

By convention, a group role does not have the LOGIN privilege. It means that you will not be able to use the group role to log in to PostgreSQL.

- To create a group role, you use the CREATE ROLE statement as follows:

```
CREATE ROLE group_role_name;
```

Example:

Step1: Create a role that represents a group without login privilege.

```
CREATE ROLE sales;
```

```
week5=> \du
```

Role name	List of roles Attributes	Member of
alice		{}
api	1000 connections	{}
bob	Cannot login	{}
dba	Create DB	{}
dev_api	Password valid until 2023-02-27 00:00:00+05:30	{}
joe		{}
john	Superuser	{}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
sales	Cannot login	{}

Step 2: To add a role to a group role, you use the following form of the GRANT statement:

```
GRANT group_role to user_role;
```

Example:

```
grant sales to alice;
```

```
week5=> \du
```

Role name	List of roles Attributes	Member of
alice		{sales}
api	1000 connections	{}
bob	Cannot login	{}
dba	Create DB	{}
dev_api	Password valid until 2023-02-27 00:00:00+05:30	{}
joe		{}
john	Superuser	{}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
sales	Cannot login	{}

- To remove a user role from a group role, you use REVOKE statement:

```
REVOKE group_role FROM user_role;
```

Example:

```
REVOKE sales FROM alice;
```

## PostgreSQL Role Membership example

A role can use privileges of the group role in the following ways:

- First, a role with the **INHERIT** attribute will automatically have privileges of the group roles of which it is the member, including any privileges inherited by that role.
- Second, a role can use the **SET ROLE** statement to temporarily become the group role. The role will have privileges of the group role rather than its original login role. Also, the objects are created by the role are owned by the group role, not the login role.

Step 1. Setting up a sample database and tables

1) Login to the PostgreSQL using the postgres database.

2) Create a new database called corp:

```
create database corp;
```

3) Switch to the corp database:

```
\c corp
```

4) Create the contacts table:

```
create table contacts(  
    id int generated always as identity primary key,  
    name varchar(255) not null,  
    phone varchar(255) not null  
);
```

5) Create the forecasts table:

```
create table forecasts(  
    year int,  
    month int,  
    amount numeric  
);
```

## Step 2. Setting roles and group roles

1) Create a role jane that can log in with a password and inherit all privileges of group roles of which it is a member:

```
CREATE ROLE jane  
INHERIT  
LOGIN  
PASSWORD 'securePass1';
```

2) Grant the select on the forecasts table to jane:

```
GRANT select
ON forecasts
TO jane;
```

3) Use the \z command to check the grant table:

```
corp=# \z
```

Schema	Name	Type	Access privileges	Column privileges	Policies
public	contacts	table			
public	contacts_id_seq	sequence			
public	forecasts	table	postgres=arwdDxt/postgres+ jane=r/postgres		

(3 rows)

4) Create the marketing group role:

```
CREATE ROLE marketing
NOINHERIT;
```

5) Create the planning group role:

```
CREATE ROLE planning
NOINHERIT;
```

6) Grant all privileges on contacts table to marketing:

```
GRANT ALL
ON contacts
TO marketing;
```

7) Grant all privileges on forecasts table to planning:

```
GRANT ALL
ON forecasts
TO planning;
```

8) Add jane as a member of marketing:

```
GRANT marketing
TO jane;
```

9) Add marketing as a member of planning:

```
GRANT planning
TO marketing;
```

### Step 3. Using the roles

1) If you connect to PostgreSQL using the role jane , you will have privileges directly granted to jane plus any privileges granted to marketing because jane **inherits** marketing's privileges:

```
psql -U jane -d corp
```

2) It'll prompt you for the jane's password.

3) The role jane can select data from the forecasts table:

```
corp=> select * from forecasts;
 year | month | amount
-----+-----+-----
(0 rows)
```

4) And insert a row into the contacts table:

```
corp=> insert into contacts(name,phone) values ('joe','044 222 883355');
INSERT 0 1
```

5) However, jane cannot insert a row into the forecasts table:

```
corp=> insert into forecasts(year,month,amount) values(2020,1,1000);
ERROR: permission denied for table forecasts
```

6) After executing the following SET ROLE statement:

```
corp=> set role planning;
SET
```

7) The role jane will have privileges granted to planning, but not the ones granted directly to jane or indirectly to marketing.

8) Now, jane can insert a row into the forecasts table:

```
corp=> insert into forecasts(year,month,amount) values(2020,1,1000);
INSERT 0 1
corp=> select * from forecasts;
 year | month | amount 
-----+-----+-----
  2020 |     1 |    1000
(1 row)
```

9) If jane attempts to select data from the contacts table, it will fail because the role planning has no privilege on the contacts table:

```
corp=> select * from contacts;
ERROR: permission denied for table contacts
```

10) To restore the original privileges of jane, you use the RESET ROLE statement:

```
corp=> reset role;
RESET
corp=> select * from contacts;
 id | name |      phone 
----+-----+-----
   1 | joe  | 044 222 883355
(1 row)
```

## PostgreSQL ALTER ROLE

PostgreSQL ALTER ROLE statement is used to modify the attributes of a role, rename a role, and change a role's session default for a configuration variable.

- **Change attributes of a role**

```
ALTER ROLE role_name [WITH] option;
```

The option can be:

- **SUPERUSER** | **NOSUPERUSER** – determine if the role is a **superuser** or not.
- **CREATEDB** | **NOCREATEDB** – allow the role to create new databases.
- **CREATEROLE** | **NOCREATEROLE** – allow the role to create or change roles.
- **INHERIT** | **NOINHERIT** – determine if the role to inherit privileges of roles of which it is a member.
- **LOGIN** | **NOLOGIN** – allow the role to log in.
- **REPLICATION** | **NOREPLICATION** – determine if the role is a replication roles.
- **BYPASSRLS** | **NOBYPASSRLS** – determine if the role to by pass a row-level security (RLS) policy.
- **CONNECTION LIMIT** limit – specify the number of concurrent connection a role can made, -1 means unlimited.
- **PASSWORD** 'password' | **PASSWORD NULL** – change the role's password.
- **VALID UNTIL** 'timestamp' – set the date and time after which the role's password is no long valid.

Example:

1. First, log in to the PostgreSQL using the postgres role.
2. Because postgres is a superuser, it can change the role alice to be a superuser:

```
ALTER ROLE alice  
SUPERUSER;
```

3. Because postgres is a superuser, it can change the role bob to be a role with login:



```
ALTER ROLE bob
LOGIN
PASSWORD 'Password6';
```

4. The following statement sets the password of the role alice to expire until the end of 2050:

```
ALTER ROLE alice
VALID UNTIL '2050-01-01'
```

```
week5=> \du
```

Role name	List of roles Attributes	Member of
alice	Superuser Password valid until 2050-01-01 00:00:00+05:30	+ {}
api	1000 connections	{}
bob		{}
dba	Create DB	{}
dev_api	Password valid until 2023-02-27 00:00:00+05:30	{}
jane		{marketing}
joe		{}
john	Superuser	{}
marketing	No inheritance, Cannot login	{planning}
planning	No inheritance, Cannot login	{}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
sales	Cannot login	{}

The following rules are applied:

- Superusers can change any of those attributes for any role.
- Roles that have the CREATEROLE attribute can change any of these attributes for only non-superusers and no-replication roles.
- Ordinal roles can only change their passwords.
  - log in to the PostgreSQL using the bob role with the password 'Password6'.
  - Alter the password.

```
week5=> alter role bob with password 'bobPassword';
ALTER ROLE
```

- **Rename roles**

To change the name of a role, you use the following form of the ALTER ROLE statement:

```
ALTER ROLE role_name  
RENAME TO new_name;
```

In this syntax, you specify the name of the role after the ALTER ROLE keywords and the new name of the role after the TO keyword.

A superuser can rename any role. A role with the CREATEROLE privilege can rename no-superuser roles.

If you use a role to log in to the PostgreSQL database server and rename it in the current session, you will get an error:

E.g. after logging in to PostgreSQL database using the role bob and then trying to rename bob to ben:

```
week5=> alter role bob rename to ben;  
ERROR: session user cannot be renamed
```

In this case, you need to connect to the PostgreSQL database server using a different role to rename that role.

You execute the following statement from the postgres' session to rename the role bob to ben

```
ALTER ROLE bob  
RENAME TO ben;
```

- **Change a role's session default for a configuration variable**

The following ALTER ROLE statement changes the role's session default for a configuration variable:

```
ALTER ROLE role_name | CURRENT_USER | SESSION_USER | ALL  
[IN DATABASE database_name]  
SET configuration_param = { value | DEFAULT }
```

In this syntax:

- First, specify the name of the role that you want to modify the role's session default, or use the CURRENT\_USER, or SESSION\_USER. You use the ALL option to change the settings for all roles.
- Second, specify a database name after the IN DATABASE keyword to change only for sessions in the named database. In case you omit the IN DATABASE clause, the change will be applied to all databases.
- Third, specify the configuration parameter and the new value in the SET clause.

Superusers can change session defaults of any roles. Roles with the CREATEROLE attribute can set the defaults for non-superuser roles. Ordinary roles can only set defaults for themselves. Only superusers can change a setting for all roles in all databases.

The following example uses the ALTER ROLE to give the role elephant a non-default, database-specific setting of the client\_min\_messages parameter:

```
ALTER ROLE elephant  
IN DATABASE dvdrental  
SET client_min_messages = NOTICE;
```

## PostgreSQL DROP ROLE statement

To remove a specified role, you use the DROP ROLE statement:

```
DROP ROLE [IF EXISTS] target_role;
```

In this syntax:

- Specify the name of the role that you want to remove after the DROP ROLE keywords.
- Use the IF EXISTS option if you want PostgreSQL to issue a notice instead of an error when you remove a role that does not exist.

To remove a superuser role, you need to be a superuser.

To drop non-superuser roles, you need to have the CREATEROLE privilege.

When you remove a role referenced in any database, PostgreSQL will raise an error. In this case, you have to take two steps:

- First, either remove the database objects owned by the role using the DROP OWNED statement or reassign the ownership of the database objects to another role REASSIGN OWNED.
- Second, revoke any permissions granted to the role.

The REASSIGN OWNED statement reassigns the ownership of all dependent objects of a target role to another role. Because the REASSIGN OWNED statement can only access objects in the current database, you need to execute this statement in each database that contains objects owned by the target role.

After transferring the ownerships of objects to another role, you need to drop any remaining objects owned by the target role by executing the DROP OWNED statement in each database that contains objects owned by the target role.

In other words, you should execute the following statements in sequence to drop a role:

```
-- execute these statements in the database that contains
-- the object owned by the target role
REASSIGN OWNED BY target_role TO another_role;
DROP OWNED BY target_role;

-- drop the role
DROP ROLE target_role;
```

Let's see the following example.

- First, we will create a new role called don and use this role to create a table named customers.
- Then, we will show you step by step how to remove the role don from the PostgreSQL database server.

## Step 1. Setting a new role and database

1. login to PostgreSQL using the postgres role:

```
psql -U postgres
```

2. create a new role called alice

```
CREATE ROLE alice
LOGIN PASSWORD 'alicePass';
```

3. grant createdb privilege to alice

```
ALTER ROLE alice
CREATEDB;
```

4. Exit the current session:

## Step 2. Using the new role to create database objects

1. login to the PostgreSQL database server using the alice role:

```
psql -U alice postgres
```

2. create a new database called sales:

```
postgres=> create database sales;  
CREATE DATABASE
```

3. Change database to sales

```
postgres=> \c sales  
You are now connected to database "sales" as user "alice".
```

4. create a new table in the sales database:

```
sales=> create table customer(customer_id serial primary key, customer_name varchar(25) not null);  
CREATE TABLE
```

5. show the table list in the sales database:

```
sales=# \dt  
List of relations  
Schema | Name | Type | Owner  
-----+-----+-----+-----  
public | customer | table | alice  
(1 row)
```

6. quit the current session:

```
postgres=# \q
```

### Step 3. Removing the role alice

1. login to the PostgreSQL database server using the postgres role:

```
psql -U postgres
```

2. Second, attempt to drop the role alice:

```
DROP ROLE alice;
```

PostgreSQL issued the following error:

```
ERROR:  role "alice" cannot be dropped because some objects depend on it  
DETAIL:  owner of database sales  
2 objects in database sales  
SQL state: 2BP01
```

The role alice cannot be dropped because it has dependent objects.

- switch to the sales database:

```
postgres=# \c sales  
You are now connected to database "sales" as user "postgres".
```

- reassign owned objects of alice to postgres:

```
sales=# reassign owned by alice to postgres;  
REASSIGN OWNED
```

- drop owned objects by alice:

```
sales=# drop owned by alice;  
DROP OWNED
```

- drop the role alice:

```
sales=# drop role alice;  
DROP ROLE
```

- list the current roles:

```
sales=# \du
```

Role name	List of roles Attributes	Member of
api	1000 connections	{}
ben		{}
dba	Create DB	{}
dev_api	Password valid until 2023-02-27 00:00:00+05:30	{}
don	Create DB	{}
jane		{marketing}
joe		{}
john	Superuser	{}
marketing	No inheritance, Cannot login	{planning}
planning	No inheritance, Cannot login	{}
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}
sales	Cannot login	{}

You will see that the role `alice` has been removed.

## Create a View (`low_salary_employees`)

Use **CREATE VIEW** to filter employees with salary < 5000.

```
CREATE VIEW low_salary_employees AS
SELECT * FROM employee WHERE salary < 5000
```

WITH CHECK OPTION;

**Why WITH CHECK OPTION?**

- It **prevents** inserting or updating rows in the view if they **don't match** the condition (salary < 5000).

## Step 4: Test Inserts & Updates

 **Successful Insert (Salary < 5000)**

**Insert a new employee into `low_salary_employees`.**

```
INSERT INTO low_salary_employees (employee_name, salary, department_id)
VALUES ('Charlie', 4000, 3);
```

**Check:**

- **Does the insert succeed?**
- **Does Charlie appear in `low_salary_employees`?**



## Failed Insert (Salary $\geq$ 5000)

Try inserting an employee with salary  $\geq$  5000.

```
INSERT INTO low_salary_employees (employee_name, salary, department_id)
VALUES ('David', 5000, 4);
```

### Expected Error:

ERROR: new row violates WITH CHECK OPTION for view "low\_salary\_employees"  
Why?

- The salary is **not less than 5000**, so WITH CHECK OPTION blocks the insert.

## Successful Update (Salary < 5000 $\rightarrow$ Another < 5000)

Modify Alice's salary within the valid range.

```
UPDATE low_salary_employees
SET salary =xxxx
WHERE employee_name = 'alice';
```

What to Verify?

- Did the update succeed?
- Does Alice still appear in low\_salary\_employees?

## Failed Update (Salary < 5000 $\rightarrow$ Salary $\geq$ 5000)

Try increasing Alice's salary above 5000.

```
UPDATE low_salary_employees
SET salary = 5500
WHERE employee_name = 'Alice';
```

### Expected Error:

ERROR: new row violates WITH CHECK OPTION for view "low\_salary\_employees"

 Why?

- WITH CHECK OPTION prevents updates that move rows **out of the view's condition**.

