Check Your Privileges

The PostgreSQL Role System

Christophe Pettus, pgexperts.com

Hi!

- I’m Christophe.
- CEO and lead consultant at PGX.
- PostgreSQL person for a long time.
- thebuild.com
- christophe.pettus@pgexperts.com

v16

What is a role, anyway?

pgexperts.com

It’s that thing you use to log in, right?

- Well, yes, that’s part of it.
- But roles are so much more!
- PostgreSQL has a very sophisticated role and privileges system.
- Let’s explore!

OK, so what *is* a **role**, then?

- 1. A “role” is an object that holds privileges, and has attributes.
  - We’ll talk about the difference between them soon.
- 2. Roles are also used to authenticate access to the database.
  - Each session has a role associated with it (which may or may not be the one that was used to log in.)
  - Authentication is a talk in itself. Another time.
- 3. Every object in the database is owned by a particular role.

OK, so, what’s a **user**?

- It’s a role with the LOGIN attribute.
- That’s it.
- That’s all.
- No, no tricks, that’s the only thing a user is.
- We’ll exclusively use the term **role** here.

I’m sure I saw something called a **group**.

- You’ll see some mentions of a “group” in the documentation.
  - Mostly in the form of obsolete commands.
- A **group** is a role.
- There’s no special separate thing called a **group**.

Roles are cluster-wide.

- Roles are global objects, not database-specific.
- Using privileges, access to particular databases can be restricted by role.
- Privileges are all database specific.
  - Just because you can select from table t in one database doesn’t mean you can select from anything in a different database.
- Remember to do a pg_dumpall to capture them: pg_dump of a single database doesn’t!

First, let's understand privileges.

- A **privilege** is an object that allows a session to perform an operation on a database object.
  - Select from a table.
  - Create a new table in a schema.
  - Call a function.
- We say that a role **"has a privilege"** if a privilege object exists in the database that grants that role that privilege.
- A session can only perform an operation if its current role has the privilege to do it.
- But there are all kinds of ways for a role to gain a privilege.

HOW CAN YOU BECOME PRIVILEGED?

1. Be a superuser.

```
if (current_role->is_superuser) {
        return TRUE;
}
```

Superusers can do anything.

- It’s not so much that it has all privileges, as it doesn’t matter what privileges it has: the answer is always, **“Sure, go ahead.”**
- You get one superuser role (`postgres`) automatically when you create a new PostgreSQL cluster.
  - You **really** should never have more than one.
- Being a superuser is an **attribute** of the role, not a **privilege** granted to the role.
  - We’ll talk about why that’s important in a bit.

2. Be the object owner.

- The role that creates a database object is its owner (unless another owner is specified at the time).
- The owner can “give away” ownership.
  - But not to just anyone: you can only give ownership to a role you can SET ROLE to (more later).
- The owner initially has all available privileges on that object.
- All can be revoked *except* the privilege to ALTER or DROP the object.

3. Do something that is granted to PUBLIC.

- PUBLIC is a pseudo-role that is built into the system.
- All roles are (in effect) “members” of PUBLIC and inherit all of its privileges.
- Anything PUBLIC can do, all roles can do.
- Not every single privilege can be granted to PUBLIC.
- Some are granted by default (which can be a surprise, which we’ll discuss later).
- Don’t confuse the **PUBLIC** role with the **public** schema.

4. Have that privilege explicitly granted.

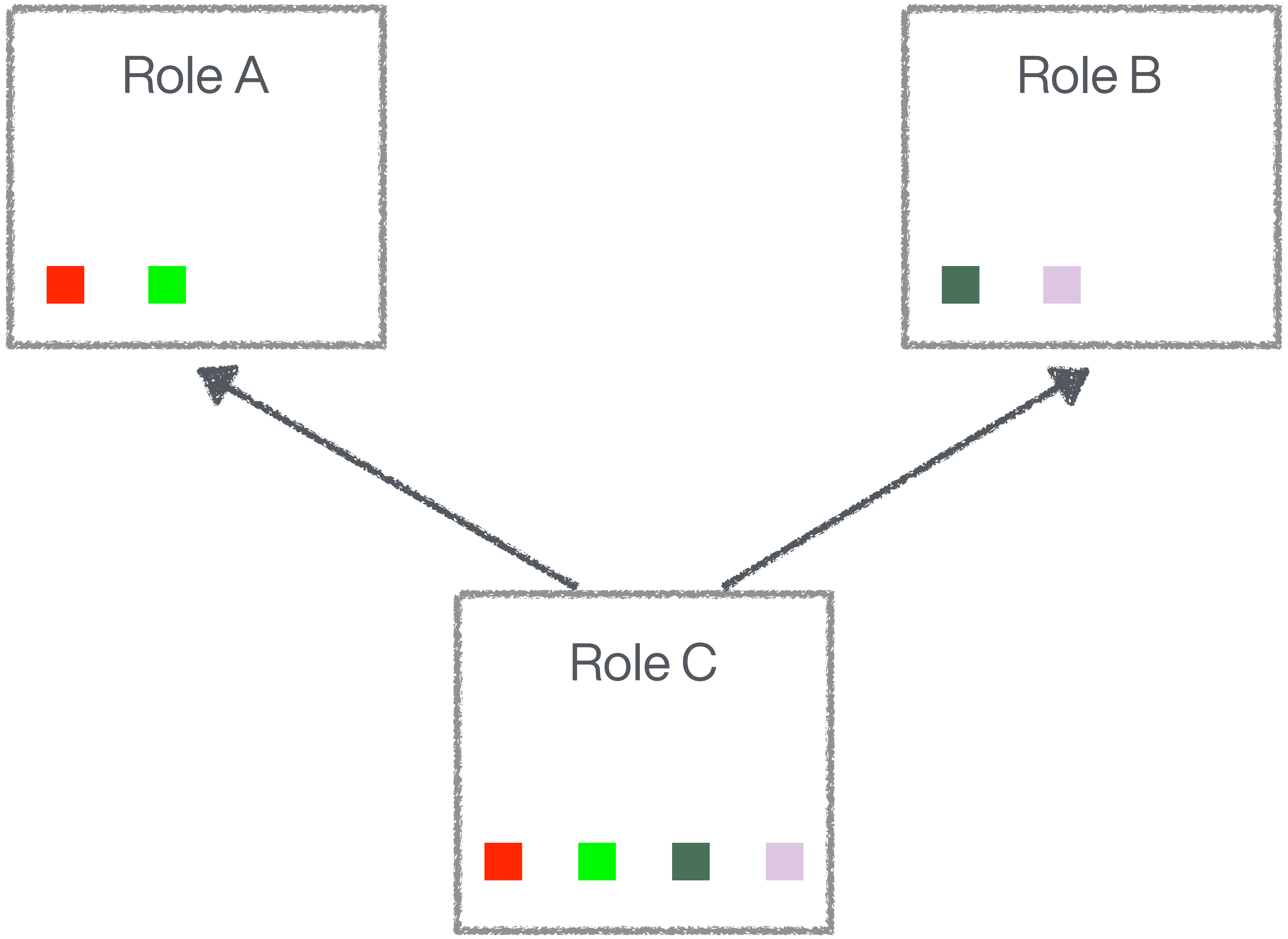
- Roles have the privileges they have been explicitly granted.
  - `GRANT SELECT ON TABLE t TO my_role;`
- Of course, what is granted can be taken away:
  - `REVOKE SELECT ON TABLE t FROM my_role;`

5. Inherit the privilege from another role.

- Roles can be members of other roles. (That’s where the “group” thing comes from.)
- A role can inherit the privileges of the roles it is a member of.
    - **Can** but not always **does**: there are controls here!
- Only **privileges** are inherited, not **attributes**.
- To fully understand how this works, let’s talk about…

Role Inheritance.

- A role can be a “member” of another role.
- This is a directed graph: one role can be a member of multiple roles.
- By default, if you don’t specify anything else, a role will inherit all of the privileges of its “parent” role.
- This is recursive, so the privileges “build up” as you work your way down the graph.
- A role can be assigned as a member of another group when it is created, when the parent is created, or later.


Role A
Role B
Role C

How do you become a member of a role?

- A role becomes a member of another role with a form of the GRANT command:
  - `GRANT <parent role> TO <child role>;`
- A role can also be added to the parent when the child is created:
  - `CREATE ROLE <child role> IN ROLE <parent role>, …;`
- Or a role can be added to the parent when the parent is created:
  - `CREATE ROLE <parent role> ROLE <child role>, …;`

Membership has its privileges.

- A grant of membership in a role can have options associated with it:
  - **SET** — This option lets a session using the child role SET ROLE to the new role.
  - **ADMIN** — This option lets the child role add and remove new members (“siblings”) to/from the parent role.
    - This is similar to **WITH GRANT OPTION** on grants of a privilege.
  - **INHERIT** — This option lets the child role inherit all of the privileges of the parent role.

Inheritance Controls.

- A role can be created with NOINHERIT: It will not inherit anything from any parent (unless you override that).
- A role can be added to another role with INHERIT FALSE: the “child” role won’t inherit anything from that particular parent role.
- Inheritance is all-or-nothing: a child gets all of the privileges of the parent, or none of them.
  - You can’t revoke an inherited privilege directly on the inheriting role.
- If a child does not inherit the privileges of its parent, its children don’t either (no generation-skipping).
- So, why be a member of a role you don’t inherit from?

6. Switch to a role that has the privilege.

- A session can change roles.
- An old role can change to a new role, if:
  - The old role is a member of the role you are changing to, **and,**
  - The old role has the SET option on the new role. (This can be granted when added to the new role, or afterwards.)
  - The SET option can come from a parent of the new role, as long as there is an unbroken chain of SETs.
- (You also need the SET option to “give away” an object that you own to a different role.)

OK, great, but how do you get privileges in the first place?

- A role is **granted** privileges by another role.
  - Using the GRANT statement, to no one's surprise.
- A superuser can grant any privilege to any role.
- A role can grant another role a privilege if it was granted that privilege WITH GRANT OPTION.
  - Role x: `GRANT SELECT ON TABLE t TO a WITH GRANT OPTION;`
  - Role a: `GRANT SELECT ON TABLE t TO b;`

Forms of GRANT.

- GRANT SELECT ON TABLE t TO role1;
- GRANT ALL PRIVILEGES ON TABLE t TO role1;
- GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA schema1 TO role1;
- GRANT SELECT ON TABLE t TO role1 WITH GRANT OPTION;
- GRANT SELECT ON TABLE t TO ROLE role1 GRANTED BY role2;

Ownership has its privileges.

- The owner of an object can grant any privilege on that object (even if it doesn't have it itself).
  - This means that the owner can "restore" to itself a privilege that has been revoked.
  - Revoking a privilege from the owner is for safety, not security.
- The ability to modify or drop an object acts like a privilege, but can't be granted. It can be inherited, though.

REVOKE.

- A superuser can revoke any privilege.
- A role can revoke any privilege that it granted (which role granted the privilege is tracked).
- If revoking a privilege on a role that has granted it other roles, you must specify CASCADE on the REVOKE statement (or you'll get an error).
- You can revoke just the WITH GRANT OPTION. Any roles that have inherited that privilege will have it revoked (assuming you specify CASCADE), but the direct role will keep it.
- The INHERIT, SET, and ADMIN options can be revoked as well.

Variations on GRANT.

- GRANT can grant a role privileges on a whole class of object at once.
  - `GRANT SELECT ON ALL TABLES IN SCHEMA my_schema TO my_role;`
- This is a one-time operation; new tables created in that schema do not automatically get the same grants.
- GRANT can also grant all privileges at once:
  - `GRANT ALL PRIVILEGES ON TABLE t TO my_role;`
- The privileges can be individually revoked after such a grant, or revoked all at once.

Grants on PUBLIC.

- PUBLIC can be granted additional privileges beyond the defaults.
- This automatically grants all roles in the system the same privileges.
- Revoking them from PUBLIC revokes them from all roles (if the role gets them from PUBLIC).
- WITH GRANT OPTION can’t be used on grants to PUBLIC, because c’mon.

GRANT IS AN OBJECT.

REVOKE IS AN OPERATION.

You can only REVOKE what was GRANTed.

- A REVOKE operation will only revoke a privilege that has been GRANTed.
- It doesn’t “block” the privilege if the object you revoked it from gets it from somewhere else.
- Think of it as:
  - GRANT creates a privilege object.
  - REVOKE deletes a privilege object, but there is no “revoke” object.
- You **may or may** not get a warning when you revoke a non-existent privilege!

```
z=> select current_user;
 current_user
--------------
 x
(1 row)

z=> CREATE FUNCTION f() RETURNS INT AS $$ SELECT 1; $$ LANGUAGE SQL;
CREATE FUNCTION
z=> SELECT f();
 f
---
 1
(1 row)

z=> REVOKE EXECUTE ON FUNCTION f() FROM x;
REVOKE
z=> SELECT f();
 f
---
 1
(1 row)
```

```
z=> REVOKE EXECUTE ON FUNCTION f() FROM y;
REVOKE
z=>
\q
Swift:~ xof$ psql -U y z;
psql (16.3)
Type "help" for help.

z=> SELECT f();
 f
---
 1
(1 row)
```

The call is coming from inside the house.

- y got its EXECUTE privilege via PUBLIC.
- PUBLIC has certain default privileges on some database objects:
    - EXECUTE on all functions and procedures.
    - CONNECT and TEMPORARY on databases.
    - USAGE on languages and data types,
- The database owner or a superuser can revoke these. (But don't unless you know what you are doing.)

Attributes.

- Roles also have **attributes** in addition to granted privileges.
- They are not GRANTed: they are assigned when the role is created, or later with ALTER ROLE.
- They are **never** inherited.

Attributes

- SUPERUSER | NOSUPERUSER
- CREATEDB | NOCREATEDB
- CREATEROLE | NOCREATEROLE
- INHERIT | NOINHERIT
- LOGIN | NOLOGIN
- REPLICATION | NOREPLICATION — Pragmatically, must have LOGIN as well.
- BYPASSRLS | NOBYPASSRLS
- CONNECTION LIMIT **connlimit**
- [ENCRYPTED] PASSWORD **'password'** | PASSWORD NULL
- VALID UNTIL **'timestamp'**
- IN ROLE role_name [, ...]
- IN GROUP role_name [, ...]
- ROLE role_name [, ...]
- ADMIN role_name [, ...]
- SET ***configuration_parameter***

SET *configuration_parameter*

- Sets the named configuration parameter when a role connects to the database.
- Does not set it on SET ROLE, which is a shame.
- Is not inherited, which is really a shame.
- Only works with configuration parameters that you can SET (“on the command line” in the documentation).

pgexperts.com

Role Administration.

- **New in version 16!** Practical role administration that does not require a superuser.
- A role with the CREATEROLE attribute can create new roles in the database.
  - It is automatically granted the ADMIN privilege on any role it creates.
  - It can be granted ADMIN on existing roles as well.
- A role with CREATEROLE (on itself) and ADMIN on its parent role can add and remove members from the parent role.
- Can’t create superuser or REPLICATION roles.

Pre-version-16.

- The ADMIN privilege doesn’t exist, so…
- A role with CREATEROLE can manipulate any role in the system, even ones it did not create.
- This allows a role with CREATEROLE to “break out” of many access controls.
  - Such as being able to access the underlying filesystem.
- Considered Harmful.

Role Playing.

- A session has two roles associated with it:
    - The **current role,** which is the role whose privileges are applied to operations. (`current_user`)
    - The **session role,** which is (usually but not always) the role that the session logged in as. (`session_user`)
    - You can also get the role used for authentication and the authentication method, and those never change during the life of the session. (`system_user`)
- Two confusingly similar ways to adopt a new role.

SET ROLE

- Changes the current role, but not the session role, to the new role.
- The old role must be a member of the new role, with the SET option. (Or the old role is a superuser.)
- You can reset back to the session role (SET ROLE NONE) or to the original authenticated role (RESET ROLE). (These are the same in 100%-ε cases.)
- Non-superusers can use this to temporarily escalate their privileges, if set up properly (example later).
- Use this one.

SET SESSION AUTHORIZATION

- Can only be used if the authenticated role is a superuser.
- Changes both the session user and the current user to the new role.
- You will probably never use this statement.

WHAT PRIVILEGES ARE THERE?

A stroll through the privilege garden.

- Each database object class has a specific set of privileges that can be granted on it.
- Often, privileges share a name but not semantics (or share semantics just conceptually).
    - USAGE on a schema isn’t the same as USAGE on a foreign data wrapper.
- Not every combination of privileges make sense.
    - Some privileges are only practical in combination with others.

Privileges on Tables.

- **SELECT** — Select from the table.
- **INSERT** — Insert into the table. (Needs UPDATE for ON CONFLICT DO UPDATE.)
- **UPDATE** — Update rows in the table. (De facto requires SELECT.)
- **DELETE** — Delete from the table. (De facto requires SELECT.)
- **TRUNCATE** — Truncate the table.
- **REFERENCES** — Create a foreign key constraint referencing (“pointing to”) this table.
- **TRIGGER** — Can create a trigger on the table. (Not required to run the trigger.)

Privileges on Tables.

- SELECT, INSERT, UPDATE, REFERENCES can be granted on individual columns instead of the entire table.
  - For INSERT, non-granted columns must have defaults or an appropriate BEFORE trigger.
  - You can't revoke access to columns individually; you need to revoke access to the whole table and re-grant.
  - Using row-level security, can be granted on a subset of rows (beyond scope of this talk).
- Can be granted to an individual table, or all tables in a schema at once.
  - Only applies to existing tables; privileges on new tables not automatically granted.
- Views and materialized use the same command syntax; you can even call them TABLEs to be confusing.

What about indexes?

- Indexes do not have separate privileges.
- Whatever a role has privileges to do on a table, it has sufficient privileges on the index.
- INSERT on a table implies the privilege to scan the index to implement a CHECK constraint.
- No current way of preventing a specific role from using an index.

Privileges on Sequences.

- **USAGE** — Allows use of `currval` and `nextval`.
- **SELECT** — Allows use of `currval`.
- **UPDATE** — Allows use of `nextval` and `setval`.
- Privileges on tables and their associated sequences are set separately. Granting INSERT on a table without USAGE on its sequences will probably result in errors.

Privileges on Schemas.

- **CREATE** — Allows creation of objects within the schema.
- **USAGE** — Allows roles to “see” objects inside of the schema.

Privileges on Databases.

- **CREATE** — Allows creation of new schemas and publications (for replication), and creating trusted extensions.
- **CONNECT** — Allows the role to connect to the database. Not much fun without the LOGIN attribute. Revoking it doesn't force-disconnect sessions using that role.
  - Automatically granted to PUBLIC, and probably not a good idea to revoke it. Use the LOGIN attribute instead.
- **TEMPORARY (or TEMP)** — Allows creation of temporary tables.
  - Automatically granted to PUBLIC, and only revoke it if you know what you are doing.

Privileges on Functions, Procedures, etc.

- **EXECUTE** — This is the only privilege available for these. Lets the role execute the function.
- Functions and procedures can be declared as SECURITY DEFINER, which means they adopt the role of the owner when running (instead of running as the invoker).
- Using ALL FUNCTIONS includes trigger functions and user-defined aggregate and window functions, but not procedures; you have to explicitly say ALL PROCEDURES for that.
- If you want to capture everything with one GRANT, you can use ALL ROUTINES.

Things that just have USAGE.

- Domains.
- Foreign data wrappers.
- Foreign servers.
- Languages.
- Types.

Exotica.

- Tablespaces just have CREATE (there’s no way of preventing a role from using a tablespace as a whole).
- Database parameters have SET (allows superuser-only parameters to be set by other roles) and ALTER SYSTEM (allows a role to issue an ALTER SYSTEM to set a parameter globally).
- Large objects have SELECT and UPDATE. Don’t use large objects.

Default Privileges.

- Setting privileges on newly-created objects can be tedious.
- ALTER DEFAULT PRIVILEGES is there for you!
- Sets the default privileges for newly-created objects system-wide, or in a particular schema, and,
- For a particular role or for all roles.

Dropping Roles.

- If a role owns objects, the system won’t let you drop it.
- Reassign ownership of all the objects the role owns to another role, then drop the role.
  - REASSIGN OWNED makes this much easier.
- Dropping a role that has members just removes the members from the role; it doesn’t drop the members.

Predefined Roles.

- PostgreSQL defines a bunch of handy roles that you can grant. Notable ones include:
  - `pg_monitor` — Allows reading the system statistics views. Usually granted to a monitoring agent.
  - `pg_read_all_data` — Can read all data, even from tables without explicit grants. Does not bypass RLS unless the role also has the BYPASSRLS attribute. Handy for a role that does pg_dump.
  - `pg_signal_backend` — Can signal another backend process to cancel a query, or to terminate.

COOKBOOK

An application-driven OLTP user.

```
# CREATE USER oltp;
# GRANT USAGE ON SCHEMA public TO oltp;
# REVOKE TEMPORARY ON DATABASE db FROM oltp;
  -- Requires that TEMPORARY be revoked from PUBLIC but granted to a parent role.
# GRANT ALL ON ALL TABLES IN SCHEMA public TO oltp;
# ALTER DEFAULT PRIVILEGES GRANT USAGE ON SCHEMAS TO oltp;
# ALTER DEFAULT PRIVILEGES GRANT SELECT, UPDATE, INSERT, DELETE, TRUNCATE
      ON TABLES TO oltp;
# ALTER ROLE oltp SET statement_timeout = ‘2 sec’;
# ALTER ROLE oltp SET work_mem = ‘64MB’;
# ALTER ROLE oltp SET idle_in_transaction_session_timeout = ‘1s’;
```

A “analyst” role.

```
# CREATE USER george_analyst;
# GRANT USAGE ON SCHEMA public TO george_analyst;
# CREATE SCHEMA workspace;
# GRANT CREATE ON SCHEMA workspace TO george_analyst;
# GRANT SELECT ON ALL TABLES IN SCHEMA public TO george_analyst;
# ALTER ROLE oltp SET statement_timeout = 0;
# ALTER ROLE oltp SET work_mem = ‘1GB’;
# ALTER ROLE oltp SET idle_in_transaction_session_timeout = ‘1m’;
```

A read-only user.

```
# CREATE USER read_only;
  -- If tables already exist, repeat for each schema.
# GRANT USAGE ON SCHEMA public TO read_only;
# GRANT SELECT ON ALL TABLES IN SCHEMA public TO read_only;
# ALTER DEFAULT PRIVILEGES GRANT USAGE ON SCHEMAS TO read_only;
# ALTER DEFAULT PRIVILEGES GRANT SELECT ON TABLES TO read_only;
```

A read-only user the easy way.

```
-- Useful if there are a lot of default privileges and objects already
# CREATE USER read_only;
# ALTER USER read_only SET default_transaction_read_only = true;
```

Create a DML-only user.

```
# CREATE USER dml_only;
# GRANT USAGE ON SCHEMA public TO dml_only;
# GRANT SELECT, UPDATE, INSERT, DELETE, TRUNCATE
      ON ALL TABLES IN SCHEMA public TO dml_only;
# ALTER DEFAULT PRIVILEGES GRANT USAGE ON SCHEMAS TO dml_only;
# ALTER DEFAULT PRIVILEGES GRANT SELECT, UPDATE, INSERT, DELETE, TRUNCATE
      ON TABLES TO dml_only;
```

If you are using role restrictions on functions…

```
# REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA public FROM PUBLIC;
# ALTER DEFAULT PRIVILEGES REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

A general DBA role.

```
# CREATE ROLE dba_user_role;
# GRANT ALL ON DATABASE my_db TO dba_user_role;
  -- If tables already exist, repeat for each schema.
# GRANT ALL ON SCHEMA public TO dba_user_role;
# GRANT ALL ON ALL TABLES IN SCHEMA public TO dba_user_role;
# GRANT pg_monitor TO dba_user_role;
# ALTER DEFAULT PRIVILEGES GRANT ALL ON SCHEMAS TO dml_only;
# ALTER DEFAULT PRIVILEGES ALL ON TABLES TO dml_only;
# ALTER DEFAULT PRIVILEGES ALL ON ROUTINES TO dml_only;
```

A “superuser”

```
# CREATE ROLE pgx_admin CREATEDB CREATEROLE BYPASSRLS NOLOGIN;
# GRANT CREATE ON my_db TO pgx_admin;
# GRANT ALL ON SCHEMA public TO pgx_admin;
# GRANT pg_read_all_data,
        pg_write_all_data,
        pg_read_all_settings,
        pg_read_all_stats,
        pg_stat_scan_tables,
        pg_monitor,
        pg_signal_backend,
        pg_checkpoint,
        pg_use_reserved_connections,
        pg_create_subscription WITH ADMIN OPTION;
# GRANT ALL ON PARAMETER <parameter>, ... TO pgx_admin WITH GRANT OPTION;
# ALTER DEFAULT PRIVILEGES GRANT ALL ON SCHEMAS TO dml_only WITH GRANT OPTION;
# ALTER DEFAULT PRIVILEGES GRANT ALL ON TABLES TO dml_only WITH GRANT OPTION;
# ALTER DEFAULT PRIVILEGES GRANT ALL ON ROUTINES TO dml_only WITH GRANT OPTION;

-- This role cannot log in, but other users can be granted the ability to set to it.
-- DO NOT use this user for regular operations.
```

Creating a user that can “sudo.”

```
# CREATE USER personal_role IN ROLE general_user_role;
# GRANT pgx_admin TO personal_role WITH INHERIT FALSE;
```

Tips.

- Don’t user the superuser for anything besides granting privileges to other roles.
    - Transfer ownership of each database to an appropriately-privileged “owning” user. That user can be used to apply migrations, but don’t use it for routine DML operations.
- Remember that any newly-created role only has what PUBLIC has, which isn’t much.
- Don’t revoke CONNECT from PUBLIC. Use the LOGIN attribute instead.
- Only revoke TEMPORARY or EXECUTE from PUBLIC if you are using to use a designed role hierarchy.

BUT ABOVE ALL...

KISS

DON'T OVER-ENGINEER YOUR ROLE SYSTEM.

PGX
pgexperts.com

QUESTIONS?

THANK YOU!

PGX

pgexperts.com