Database Roles

- Database Roles
- Role Attributes
- Role Membership
- Dropping Roles
- Predefined Roles
- Function Security

PostgreSQL manages database access permissions using the concept of roles. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables and functions) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant membership in a role to another role, thus allowing the member role to use privileges assigned to another role.

The concept of roles subsumes the concepts of “users” and “groups”. In PostgreSQL versions before 8.1, users and groups were distinct kinds of entities, but now there are only roles. Any role can act as a user, a group, or both.

This chapter describes how to create and manage roles. More information about the effects of role privileges on various database objects can be found in ???.

Database Roles

role

user

CREATE ROLE

DROP ROLE

Database roles are conceptually completely separate from operating system users. In practice it might be convenient to maintain a correspondence, but this is not required. Database roles are

global across a database cluster installation (and not per individual database). To create a role use the `CREATE ROLE` SQL command: CREATE ROLE <name>; <name> follows the rules for SQL identifiers: either unadorned without special characters, or double-quoted. (In practice, you will usually want to add additional options, such as `LOGIN`, to the command. More details appear below.) To remove an existing role, use the analogous `DROP ROLE` command: DROP ROLE <name>;

createuser

dropuser

For convenience, the programs ??? and ??? are provided as wrappers around these SQL commands that can be called from the shell command line: createuser <name> dropuser <name>

To determine the set of existing roles, examine the pg_roles system catalog, for example: SELECT rolname FROM pg_roles; or to see just those capable of logging in: SELECT rolname FROM pg_roles WHERE rolcanlogin; The ??? program's `\du` meta-command is also useful for listing the existing roles.

In order to bootstrap the database system, a freshly initialized system always contains one predefined login-capable role. This role is always a “superuser”, and it will have the same name as the operating system user that initialized the database cluster with `initdb` unless a different name is specified. This role is often named `postgres`. In order to create more roles you first have to connect as this initial role.

Every connection to the database server is made using the name of some particular role, and this role determines the initial access privileges for commands issued in that connection. The role name to use for a particular database connection is indicated by the client that is initiating the connection request in an application-specific fashion. For example, the `psql` program uses the `-U` command line option to indicate the role to connect as. Many applications assume the name of the current operating system user by default (including `createuser` and `psql`). Therefore it is often convenient to maintain a naming correspondence between roles and operating system users.

The set of database roles a given client connection can connect as is determined by the client authentication setup, as explained in ???. (Thus, a client is not limited to connect as the role matching its operating system user, just as a person's login name need not match his or her real name.) Since the role identity determines the set of privileges available to a connected client, it is important to carefully configure privileges when setting up a multiuser environment.

Role Attributes

A database role can have a number of attributes that define its privileges and interact with the client authentication system.

login privilege
Only roles that have the `LOGIN` attribute can be used as the initial role name for a database connection. A role with the `LOGIN` attribute can be considered the same as a “database user”. To create a role with login privilege, use either:

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

(`CREATE USER` is equivalent to `CREATE ROLE` except that `CREATE USER` includes `LOGIN` by default, while `CREATE ROLE` does not.)

superuser status
A database superuser bypasses all permission checks, except the right to log in. This is a dangerous privilege and should not be used carelessly; it is best to do most of your work as a role that is not a superuser. To create a new database superuser, use `CREATE ROLE name SUPERUSER`. You must do this as a role that is already a superuser.

database creation
A role must be explicitly given permission to create databases (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name CREATEDB`.

role creation
A role must be explicitly given permission to create more roles (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name CREATEROLE`. A role with `CREATEROLE` privilege can alter and drop roles which have been granted to the `CREATEROLE` user with the `ADMIN` option. Such a grant occurs automatically when a `CREATEROLE` user that is not a superuser creates a new role, so that by default, a `CREATEROLE` user can alter and drop the roles which they have created. Altering a role includes most changes that can be made using `ALTER ROLE` , including, for example, changing passwords. It also includes modifications to a role that can be made using the `COMMENT` and `SECURITY LABEL` commands.

However, `CREATEROLE` does not convey the ability to create `SUPERUSER` roles, nor does it convey any power over `SUPERUSER` roles that already exist. Furthermore, `CREATEROLE` does not convey the power to create `REPLICATION` users, nor the ability to grant or revoke the `REPLICATION` privilege, nor the ability to modify the role properties of such users. However, it does allow `ALTER ROLE ... SET` and `ALTER ROLE ... RENAME` to be used on `REPLICATION` roles, as well as the use of `COMMENT ON ROLE`, `SECURITY LABEL ON ROLE`, and `DROP ROLE`. Finally, `CREATEROLE` does not confer the ability to grant or revoke the `BYPASSRLS` privilege.

initiating replication
A role must explicitly be given permission to initiate streaming replication (except for superusers, since those bypass all permission checks). A role used for streaming replication must have `LOGIN` permission as well. To create such a role, use `CREATE ROLE name REPLICATION LOGIN`.

password
A password is only significant if the client authentication method requires the user to supply a

password when connecting to the database. The `password` and `md5` authentication methods make use of passwords. Database passwords are separate from operating system passwords. Specify a password upon role creation with `CREATE ROLE name PASSWORD 'string'`.

inheritance of privileges
A role inherits the privileges of roles it is a member of, by default. However, to create a role which does not inherit privileges by default, use `CREATE ROLE name NOINHERIT`. Alternatively, inheritance can be overridden for individual grants by using `WITH INHERIT TRUE` or `WITH INHERIT FALSE`.

bypassing row-level security
A role must be explicitly given permission to bypass every row-level security (RLS) policy (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name BYPASSRLS` as a superuser.

connection limit
Connection limit can specify how many concurrent connections a role can make. -1 (the default) means no limit. Specify connection limit upon role creation with `CREATE ROLE name CONNECTION LIMIT 'integer'`.

A role's attributes can be modified after creation with `ALTER ROLE`. See the reference pages for the ??? and ??? commands for details.

A role can also have role-specific defaults for many of the run-time configuration settings described in ???. For example, if for some reason you want to disable index scans (hint: not a good idea) anytime you connect, you can use:

```
ALTER ROLE myname SET enable_indexscan TO off;
```

This will save the setting (but not set it immediately). In subsequent connections by this role it will appear as though `SET enable_indexscan TO off` had been executed just before the session started. You can still alter this setting during the session; it will only be the default. To remove a role-specific default setting, use `ALTER ROLE rolename RESET varname`. Note that role-specific defaults attached to roles without `LOGIN` privilege are fairly useless, since they will never be invoked.

When a non-superuser creates a role using the `CREATEROLE` privilege, the created role is automatically granted back to the creating user, just as if the bootstrap superuser had executed the command `GRANT created_user TO creating_user WITH ADMIN TRUE, SET FALSE, INHERIT FALSE`. Since a `CREATEROLE` user can only exercise special privileges with regard to an existing role if they have `ADMIN OPTION` on it, this grant is just sufficient to allow a `CREATEROLE` user to administer the roles they created. However, because it is created with `INHERIT FALSE, SET FALSE`, the `CREATEROLE` user doesn't inherit the privileges of the created role, nor can it access the privileges of that role using `SET ROLE`. However, since any user who has `ADMIN OPTION` on a role can grant membership in that role to any other user, the `CREATEROLE` user can gain access to the created role by simply granting that role back to themselves with the `INHERIT` and/or `SET` options. Thus, the fact that privileges are not inherited by default nor is `SET ROLE` granted by default is a safeguard against accidents, not a

security feature. Also note that, because this automatic grant is granted by the bootstrap superuser, it cannot be removed or changed by the `CREATEROLE` user; however, any superuser could revoke it, modify it, and/or issue additional such grants to other `CREATEROLE` users. Whichever `CREATEROLE` users have `ADMIN OPTION` on a role at any given time can administer it.

Role Membership

role

membership in

It is frequently convenient to group users together to ease management of privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In PostgreSQL this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

To set up a group role, first create the role: CREATE ROLE <name>; Typically a role being used as a group would not have the `LOGIN` attribute, though you can set it if you wish.

Once the group role exists, you can add and remove members using the `GRANT` and `REVOKE` commands: GRANT <group_role> TO <role1>, ... ; REVOKE <group_role> FROM <role1>, ... ; You can grant membership to other group roles, too (since there isn't really any distinction between group roles and non-group roles). The database will not let you set up circular membership loops. Also, it is not permitted to grant membership in a role to `PUBLIC`.

The members of a group role can use the privileges of the role in two ways. First, member roles that have been granted membership with the `SET` option can do `SET ROLE` to temporarily “become” the group role. In this state, the database session has access to the privileges of the group role rather than the original login role, and any database objects created are considered owned by the group role, not the login role. Second, member roles that have been granted membership with the `INHERIT` option automatically have use of the privileges of those directly or indirectly a member of, though the chain stops at memberships lacking the inherit option. As an example, suppose we have done:

```
CREATE ROLE joe LOGIN;
CREATE ROLE admin;
CREATE ROLE wheel;
CREATE ROLE island;
GRANT admin TO joe WITH INHERIT TRUE;
GRANT wheel TO admin WITH INHERIT FALSE;
GRANT island TO joe WITH INHERIT TRUE, SET FALSE;
```

Immediately after connecting as role `joe`, a database session will have use of privileges granted directly to `joe` plus any privileges granted to `admin` and `island`, because `joe` “inherits” those privileges. However, privileges granted to `wheel` are not available, because even though `joe` is indirectly a member of `wheel`, the membership is via `admin` which was granted using `WITH INHERIT FALSE`. After:

```
SET ROLE admin;
```

the session would have use of only those privileges granted to `admin`, and not those granted to `joe` or `island`. After:

```
SET ROLE wheel;
```

the session would have use of only those privileges granted to `wheel`, and not those granted to either `joe` or `admin`. The original privilege state can be restored with any of:

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

> [!NOTE] The `SET ROLE` command always allows selecting any role that the original login role is directly or indirectly a member of, provided that there is a chain of membership grants each of which has `SET TRUE` (which is the default). Thus, in the above example, it is not necessary to become `admin` before becoming `wheel`. On the other hand, it is not possible to become `island` at all; `joe` can only access those privileges via inheritance.

> [!NOTE] In the SQL standard, there is a clear distinction between users and roles, and users do not automatically inherit privileges while roles do. This behavior can be obtained in PostgreSQL by giving roles being used as SQL roles the `INHERIT` attribute, while giving roles being used as SQL users the `NOINHERIT` attribute. However, PostgreSQL defaults to giving all roles the `INHERIT` attribute, for backward compatibility with pre-8.1 releases in which users always had use of permissions granted to groups they were members of.

The role attributes `LOGIN`, `SUPERUSER`, `CREATEDB`, and `CREATEROLE` can be thought of as special privileges, but they are never inherited as ordinary privileges on database objects are. You must actually `SET ROLE` to a specific role having one of these attributes in order to make use of the attribute. Continuing the above example, we might choose to grant `CREATEDB` and `CREATEROLE` to the `admin` role. Then a session connecting as role `joe` would not have these privileges

immediately, only after doing `SET ROLE admin`.

To destroy a group role, use `DROP ROLE`: DROP ROLE <name>; Any memberships in the group role are automatically revoked (but the member roles are not otherwise affected).

Dropping Roles

Because roles can own database objects and can hold privileges to access other objects, dropping a role is often not just a matter of a quick `DROP ROLE`. Any objects owned by the role must first be dropped or reassigned to other owners; and any permissions granted to the role must be revoked.

Ownership of objects can be transferred one at a time using `ALTER` commands, for example:

```
ALTER TABLE bobs_table OWNER TO alice;
```

Alternatively, the `REASSIGN OWNED` command can be used to reassign ownership of all objects owned by the role-to-be-dropped to a single other role. Because `REASSIGN OWNED` cannot access objects in other databases, it is necessary to run it in each database that contains objects owned by the role. (Note that the first such `REASSIGN OWNED` will change the ownership of any shared-across-databases objects, that is databases or tablespaces, that are owned by the role-to-be-dropped. Subscriptions, while stored in a shared catalog, belong to a specific database and are reassigned only when `REASSIGN OWNED` is executed in that database.)

Once any valuable objects have been transferred to new owners, any remaining objects owned by the role-to-be-dropped can be dropped with the `DROP OWNED` command. Again, this command cannot access objects in other databases, so it is necessary to run it in each database that contains objects owned by the role. Also, `DROP OWNED` will not drop entire databases or tablespaces, so it is necessary to do that manually if the role owns any databases or tablespaces that have not been transferred to new owners.

`DROP OWNED` also takes care of removing any privileges granted to the target role for objects that do not belong to it. Because `REASSIGN OWNED` does not touch such objects, it's typically necessary to run both `REASSIGN OWNED` and `DROP OWNED` (in that order!) to fully remove the dependencies of a role to be dropped.

In short then, the most general recipe for removing a role that has been used to own objects is:

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;
-- repeat the above commands in each database of the cluster
DROP ROLE doomed_role;
```

When not all owned objects are to be transferred to the same successor owner, it's best to handle the exceptions manually and then perform the above steps to mop up.

If `DROP ROLE` is attempted while dependent objects still remain, it will issue messages identifying which objects need to be reassigned or dropped.

Predefined Roles

role

PostgreSQL provides a set of predefined roles that provide access to certain, commonly needed, privileged capabilities and information. Administrators (including roles that have the `CREATEROLE` privilege) can `GRANT` these roles to users and/or other roles in their environment, providing those users with access to the specified capabilities and information. For example:

```
GRANT pg_signal_backend TO admin_user;
```

> [!WARNING] Care should be taken when granting these roles to ensure they are only used where needed and with the understanding that these roles grant access to privileged information.

The predefined roles are described below. Note that the specific permissions for each of the roles may change in the future as additional capabilities are added. Administrators should monitor the release notes for changes.

`pg_checkpoint`
`pg_checkpoint` allows executing the `CHECKPOINT` command.

`pg_create_subscription`
`pg_create_subscription` allows users with `CREATE` permission on the database to issue `CREATE SUBSCRIPTION`.

`pg_database_owner`
`pg_database_owner` always has exactly one implicit member: the current database owner. It cannot be granted membership in any role, and no role can be granted membership in `pg_database_owner`. However, like any other role, it can own objects and receive grants of access privileges. Consequently, once `pg_database_owner` has rights within a template database, each owner of a database instantiated from that template will possess those rights. Initially, this role owns the `public` schema, so each database owner governs local use of that schema.

`pg_maintain`
`pg_maintain` allows executing `VACUUM`, `ANALYZE`, `CLUSTER`, `REPACK`, `REFRESH MATERIALIZED VIEW`, `REINDEX`, and `LOCK TABLE` on all relations, as if having `MAINTAIN` rights on those objects.

`pg_monitor`; `pg_read_all_settings`; `pg_read_all_stats`; `pg_stat_scan_tables`
These roles are intended to allow administrators to easily configure a role for the purpose of monitoring the database server. They grant a set of common privileges allowing the role to read various useful configuration settings, statistics, and other system information normally restricted to superusers.

`pg_monitor` allows reading/executing various monitoring views and functions. This role is a member of `pg_read_all_settings`, `pg_read_all_stats` and `pg_stat_scan_tables`.

`pg_read_all_settings` allows reading all configuration variables, even those normally visible only to superusers.

`pg_read_all_stats` allows reading all pg_stat_* views and use various statistics related extensions, even those normally visible only to superusers.

`pg_stat_scan_tables` allows executing monitoring functions that may take `ACCESS SHARE` locks on tables, potentially for a long time (e.g., `pgrowlocks(text)` in the ??? extension).

`pg_read_all_data`; `pg_write_all_data`
`pg_read_all_data` allows reading all data (tables, views, sequences, large objects), as if having `SELECT` rights on those objects and `USAGE` rights on all schemas. This role does not bypass row-level security (RLS) policies. If RLS is being used, an administrator may wish to set `BYPASSRLS` on roles which this role is granted to.

`pg_write_all_data` allows writing all data (tables, views, sequences, large objects), as if having `INSERT`, `UPDATE`, and `DELETE` rights on those objects and `USAGE` rights on all schemas. This role does not bypass row-level security (RLS) policies. If RLS is being used, an administrator may wish to set `BYPASSRLS` on roles which this role is granted to.

`pg_read_server_files`; `pg_write_server_files`; `pg_execute_server_program`
These roles are intended to allow administrators to have trusted, but non-superuser, roles which are able to access files and run programs on the database server as the user the database runs as. They bypass all database-level permission checks when accessing files directly and they could be used to gain superuser-level access. Therefore, great care should be taken when granting these roles to users.

`pg_read_server_files` allows reading files from any location the database can access on the server using `COPY` and other file-access functions.

`pg_write_server_files` allows writing to files in any location the database can access on the server using `COPY` and other file-access functions.

`pg_execute_server_program` allows executing programs on the database server as the user the database runs as using `COPY` and other functions which allow executing a server-side program.

`pg_signal_autovacuum_worker`
`pg_signal_autovacuum_worker` allows signaling autovacuum workers to cancel the current table's vacuum or terminate its session. See ???.

`pg_signal_backend`
`pg_signal_backend` allows signaling another backend to cancel a query or terminate its session. Note that this role does not permit signaling backends owned by a superuser. See ???.

`pg_use_reserved_connections`
`pg_use_reserved_connections` allows use of connection slots reserved via ???.

Function Security

Functions, triggers and row-level security policies allow users to insert code into the backend server that other users might execute unintentionally. Hence, these mechanisms permit users to “Trojan horse” others with relative ease. The strongest protection is tight control over who can define objects. Where that is infeasible, write queries referring only to objects having trusted owners. Remove from `search_path` any schemas that permit untrusted users to create objects.

Functions run inside the backend server process with the operating system permissions of the database server daemon. If the programming language used for the function allows unchecked memory accesses, it is possible to change the server's internal data structures. Hence, among many other things, such functions can circumvent any system access controls. Function languages that allow such access are considered “untrusted”, and PostgreSQL allows only superusers to create functions written in those languages.

□□□ #2
□ □□□ □□□ 2026-10-31 14:15:08 UTC
□ □□□ □□□ 2026-10-31 14:15:09 UTC