Users, Groups, Roles and Privileges

Max Ganz II @ Redshift Research Project

1st February 2023 (updated 20th May 2024)

Abstract

In Postgres, roles replaced users and groups (both became roles). Roles in Redshift are different, and do not replace users or groups but exist alongside them as a third, separate, first-class type. Both roles and groups can be granted privileges, with roles then being granted to users, or users being added to groups, these two operations being identical in effect. Roles however can also be granted to roles, allowing a hierarchy of roles to be constructed. Along with roles come a new set of Redshift-specific privileges, which can be granted to and only to roles (roles can also be granted the existing Postgres privileges). These new privileges are unlike the existing Postgres privileges (which are per-user, per-object) as they are per-user only; a user, once holding one of the new privileges (which must be via a role), holds that privilege everywhere, always, in all databases; the new privileges are essentially fragments of the super user. Finally, note the documentation for roles is particularly egregious, and that there are a few of the new privileges which are properly tantamount to granting super user, as they allow a user to elevate themselves to super user.

Contents

Introduction

Redshift is derived from Postgres and by this inherited the Postgres privileges mechanism.

The only change made to this mechanism occurred recently (in about 2023, IIRC), where a `drop` privilege was added.

Roles have now been added to Redshift, but they are *not* Postgres roles - `pg_roles` doesn't exist, for example, and so the name is rather misleading. Think of them as "Redshift roles", and start with a blank slate.

In Postgres, roles superseded and replaced both users and groups. Users and groups under the hood both became roles, with the existing commands for users and groups under the hood now in fact making roles (and the docs being clear that this is the case).

This is *not* the case in Redshift. Roles in Redshift are not a port or implementation of roles in Postgres; Redshift now has users, groups *and* a new mechanism named "roles", all as separate, co-existing and interacting first-class entities, and, critically, roles have in fact introduced a separate, second privileges mechanism, entirely different in its nature and implementation to the original Postgres mechanism.

This document then examines, describes and compares, in detail, the existing Postgres privileges mechanism, the new Redshift privileges mechanism, and their interactions.

Privilege Mechanisms

There are in Redshift a number of different types of object - databases, schemas, tables, functions, and so on.

Each object behaves and is used in certain ways; for example, you can issue `SELECT` on a table but not on a database, so a privilege to permit or refuse `SELECT` exists for (and only makes sense for) a table.

We find then naturally emerging from the set of different types of objects a set of privileges, where privileges make sense typically for only one or two types of object (you might well think it would only ever be that a given privilege made sense for one type of object, but in Redshift (and Postgres) there are quite often pairs of objects which are very similar in their nature - tables and views, and functions and procedures, for example - such that they share most of their privileges).

Now, in Postgres (up to version 8.1), and so also in Redshift (as it derived from Postgres 8.0), privileges are arranged on the basis of a privilege for each action, which is assigned per object, per user; *user Frodo has privilege usage on schema ring.*

To help with organizing privileges, there exists also the concept of *groups.* Users can belong to groups, and both users and groups can hold privileges. A user held their own privileges, plus those of any group they were a member of. Groups could not be members of groups.

With Postgres 8.1, users and groups were done away with, and there were then *only* roles. A user is really a role; a group is really a role. The only property that makes a user special is that it's a role which is allowed to log into the server. So now it's roles, all the way down - a role can belong to a role, a role can hold privileges, and a role holds the privileges of any roles it belongs to. All the existing operations on users and groups, although they are for compatibility still supported, actually now perform operations on roles, and this is explained in the documentation.

This is lovely - pure, simple, utterly flexible, easy to reason about.

We now however find something also and unfortunately called "roles" in Redshift.

As has been stated, these are *not* Postgres roles. To my eye, what we see here is what I think we often see in Redshift, which is implementation constrained

by a large, legacy code base : in other words, something big and new bolted on the side rather than integrated into the whole (and then presented with a hefty dose of "look at our amazing new functionality" hype from AWS, combined with docs which tell you approximately nothing over the course of about 50 pages and leave everything to the imagination).

In Redshift, we still have users and groups, and they remain separate, first-class types in the database, but now we have roles *as well*. Users are *not* roles, groups are *not* roles, the Postgres system tables for roles have *not* turned up in the system tables; there is no `pg_roles`. There *is* now a `pg_role`, but it's different to the Postgres system table for roles, and there are a couple of extra, non-Postgres tables, to go along with it.

Roles in Redshift are a collection of privileges, just like a group, and roles can be granted to roles and - and this is actually the biggie - there are a set of *new* privileges, which can *only* be granted to roles; not to users, not to groups, and these new privileges are in a critical ways fundamentally unlike existing privileges.

Really, roles in Redshift are actually about the new privileges and this new privileges mechanism. The "role" part of this is pretty much inconsequential : in terms of organizing privileges, a role is a group, excepting that a role can also be granted to a role. That's it.

From now on, the previously existing privileges I'll call Postgres privileges, or Postgres-style privileges, and the new privileges I'll call Redshift privileges or Redshift-style privileges.

For a user to obtain a Redshift privilege, the privilege must be granted to a role, and that role granted to the user. Roles cannot be granted to groups.

Redshift privileges, in stark contrast to Postgres privileges, are *global*; when a user is granted (which must be via a role) a Redshift privilege, that privilege is always active, on all objects affected by that privilege, in all databases, *always*, regardless of anything.

With Postgres privileges, grants are per action (the privilege granted), per object (database, table, function, what-have-you), per user (directly or via group membership).

With Redshift privileges, grants are per action (the privilege granted), per user (not directly but via roles only), and *that's it*.

So for example there is a Redshift privilege, `CREATE TABLE`. If we grant this to a role, and the role we then grant to user Gandalf, user Gandalf can now create tables in all schemas, in all databases, everywhere, always and regardless of anything else.

These new privileges are essentially fragments of the super user.

Finally, note roles can also be granted Postgres privileges in the usual way, just as if they were groups.

What this means in principle (but quite possibly not in practise, as we will see, due to implementation concerns) is that roles have superseded groups; every-

thing you do with groups you can do with roles, and roles give you something new (the new Redshift privileges).

Users

In Redshift (and Postgres), users are a cluster level concept, not a database level concept; you do not get a new set of users per database. There is and only is one set of users, no matter how many databases you have.

Users, under the hood, are uniquely identified by their user ID, which is an `int4`. This increments when a new user is made, and so every user always has a unique ID.

However, from the point of view of a human using Redshift, users are uniquely identified by a name, which is an arbitrary string, which is up to 127 bytes (and note that's bytes, not characters) of `UTF-8`. All users must at any given time have a unique name, but names which were used in the past but are no longer in use, can be re-used.

Note however there are places in the system tables where users are referred to by name only - their user ID is *not* provided - which is a problem for historical data (the user name could have been re-used), and what's more, it's often the case that the full user name is not given - for example, `stv_sessions` indicates the user by the first 50 bytes only of the user name.

As such, in the Redshift system tables, to my knowledge, 50 bytes is the longest safe user name; anything more and you have the potential to run into problems where you cannot always know which user is being referred to in some system tables.

(This issues crops up with particular strength and pervasiveness for database names; often database names, which are also in principle 127 bytes (not characters) of `UTF-8`, have their first 32 bytes only stored.)

When you come to delete a user, Redshift (like Postgres) will *not* allow the user to be deleted unless that user is utterly bare of everything - owns no object, holds *no* privileges, nothing.

For the user to be deleted there can be *no* traces of that user in the database *at all* and the `drop user` command is not able to perform any of this work - it's all down to you. This necessity to rid the database completely of a user, prior to dropping the user, is how Redshift/Postgres ensures there's no confusion whereby something, somewhere, still references the deleted user's user ID.

It's also a moderate to large pain in the neck for you, when you come to delete a user, you need to go through the system table for every type of object (schemas, tables, functions, etc) and look to see if anything in there is owned by that user -

and in fact, you need to enumerate all their privileges and this something which until very recently Redshift could not do, which made the situation problematic; you needed to drop a user, but the user holds privileges, but you couldn't find out what those privileges are. Crazy, and it was like that for ten years.

As such, what I often saw was that users were *not* deleted, but rather simply have their account modified so they can no longer log in (and their password changed for good measure). This is not good practise and in time leads to confusion and problems.

There is in Postgres (and so in Redshift) the concept of *ownership* for objects (tables, views, functions, what-have-you). Each object has one and only one owner, which is a user (groups and roles cannot own objects). When a user creates an object, that user is automatically the owner of that object. Ownership of an object can be changed, but it always begins as the creating user.

The owner of an object completely bypasses the privileges mechanism, with regard to that object only.

The owner of an object can then revoke from themselves every single privilege that exists upon an object, and still then be able to perform every possible action with that object, because the privileges the owner holds are completely irrelevant as they are not being examined in any way whatsoever for the owner of the object.

Now, in Redshift there are three *types* of user; normal users, super users, and the root user, where there is a single root user, which has the name `rdsdb`, which is owned and operated by AWS.

The root user is God, and can do anything - you might have been thinking that the super users are God and can do anything, but this is in fact not the case; super users are minor deities only.

All of the built-in system objects, so the system tables, all the functions Redshift comes with, and also a lot of the background queries Redshift constantly runs, all are owned and issued by `rdsdb`, the root user, and so you cannot touch them, even if you are a super user.

(It's like Android. You do not have root on your phone. On Redshift, you also do not have root.)

A lot of the system tables are like this - access is only permitted through views, which control and constrain what you can see. This is a problem in some cases - for example, the only way to know if a table has the `auto` sort-type is by looking at `svv_table_info`, which is a big, costly view (9.5kb of text) which I expect to be buggy, and that view, last I looked, showed information only for tables which had one or more rows; it did not show information for empty tables. Nothing you can do to work around that problem, because root owns the actual tables under the view, and you're not allowed to look at them directly (which to my eye, seems like hubris; as if the devs would never make mistakes).

Moving on, we next come to super users, who normal users in every respect except one; they *always* bypass the privileges mechanism, for all objects. Super users can perform any operation, on anything, always - privileges are not being

examined - except for objects owned by `rdsdb`, which cannot be touched in any way, shape or form.

Finally, normal users are the plebs working the fields. Normal users are limited in their operations by the privileges mechanism, except for objects they themselves own. On objects they own, normal users are the same as super users; they can perform any operation, always.

Groups

In Redshift (and Postgres), groups are a cluster level concept, not a database level concept; you do not get a new set of groups per database. There is and only is one set of groups, no matter how many databases you have.

Groups, under the hood, are uniquely identified by their group ID, which is an `int4`. This increments when a new group is made, and so every group always has a unique ID.

From the point of view of a human using Redshift, groups are uniquely identified by a name, an arbitrary string, which is up to 127 bytes (not characters) of `UTF-8`, and all groups must at any given time have a unique name, but names which were used in the past but are no longer in use, can be re-used.

Groups are a pretty superficial concept in Redshift (and Postgres). The sole purpose of groups is to make arranging and organizing privileges easier.

Groups can be granted privileges, and users can belong to groups, and a user receives the privileges of a group if that user is a member of that group (which can lead to a user can receiving the same privilege on the same object more than once, which does no harm).

The only place in the system tables that you ever see groups, either by their ID or name, is in the system table `pg_group`, which lists groups and their members, and in the `ACL` columns, which are found in the system tables which describe objects to which privileges apply - such as tables, functions, databases, etc - and describe for each object which privileges are granted to who (and here the "who" can be a group name).

The `pg_group` system table has one row per group, and in that row uses an array to record the set of users in that group. Arrays are leader node only functionality and so this table - if you use the membership column - is leader node only. As it is, this table has only three columns; the group ID, group name, and membership array, so it's hard *not* to use the membership array.

The fact that group membership info is leader node only is why you see so little information available about Redshift on a per-group basis; because it's impossible to issue queries which link up group membership with the system tables which carry worker node information (as those tables typically use worker nodes).

Groups cannot own objects; only users own objects.

Groups cannot be members of groups; only users can be a member of a group.

So a group then is simply a set of privileges.

In the usual case, in real world systems, you find many users need the same set of privileges - maybe there are many BI developers, or a number of cluster admin, or what-have-you; there are distinct sets of users which are identical in the privileges they need.

Standard best practise is to *never* grant privileges to users, as this ends up being high maintenance and error prone (particularly so prior to the capability to see the privileges granted to any given user or group), but instead to create one group per distinct set of users, grant those groups the privileges needed by their set of users, and add or remove users to and from groups, in accordance to the work the users are doing.

In short, groups get privileges, users get groups. Users never get privileges.

The Group-like Object Public

Public is not a group.

If you look in `pg_group`, you will not find a group named `public`, and as such, `public` has no set of members.

Every user is automatically, always and irrevocably a member of what I call "the group-like object `public`". You *cannot* remove a user from the group-like object public; there *is* no group `public`.

You can grant privileges to `public`, by using its name where you would use a group name, and by doing so, you grant those privileges to every user in the cluster (because all users are always part of the group-like object `public`).

In system tables which describe objects to which privileges apply, such as tables and schemas, you will find an `ACL` column (access control list), which describes for each object which privileges are granted to who (I try not to write "whom", as I consider it obsolete grammar), and here the "who" can be the name `public`, and this is the and the only place the name `public` is found.

Now, Redshift (and Postgres) ship with thousands of functions, many of which are an inherent part of SQL, such as `min()` and `max()`, the type conversion functions (which we normally access via the `::` operator but which are in fact functions being called under the hood), all of the PostGIS functionality Redshift implements, and so on and so on.

All of these functions are available for everyone to use as execute privileges on functions are granted to public.

I have heard of systems where admin, I believe for security purposes, want to delete `public`. You cannot directly do this - `public` it is not a group, but an inherent part of Redshift/Postgres, so there is nothing for you to delete - the nearest you can get is to revoke all privileges from `public` from all objects, which has the effect of making the automatic membership of public wholly without effect.

Actually doing this is problematic (I'd go further and say nightmarish).

Firstly, until recently, there has been no viable method by which to enumerate the privileges granted to `public`. If you can't know what's been granted, you

can't know what to remove, or check that all privileges have in fact been removed. That alone makes it impossible.

Secondly, if you do this, and revoke all privileges granted to `public`, you will make it impossible for people to use SQL, because so much functionality will have been removed from them. You would need to figure out and grant thousands of privileges to built-in functions to a group of your own, to which you add users, so that users can use functions essential to writing SQL, which is essentially the same as leaving `public` as it is.

One final note.

When any user creates a function or a procedure (internally, Redshift/Postgres pretty much thinks these are the same thing - there is only one system table, `pg_proc`, for both), the `execute` privilege on that function or procedure is automatically granted to `public`.

You can't stop this, so if you are trying to keep `public` stripped of grants, you'd also need to remove these automatically granted privileges on an ongoing basis.

Roles

Roles in Redshift are not the same as roles in Postgres.

In Postgres, it used to be (up to version 8.1, and remember here Redshift derives from 8.0) there were users and groups, and these were separate, first-class types in the database. Users could belong to groups, and both users and groups could hold privileges. A user held their own privileges, plus those of any group they were a member of. Groups could not be members of groups.

With Postgres 8.1, users and groups were done away with, and in Postgres there were *only* roles. A user is really a role; a group is really a role. The only property that makes a user special is that it's a role which is allowed to log into the server. So now it's roles, all the way down - a role can belong to a role, a role can hold privileges, and a role holds the privileges of any roles it belongs to. All the existing operations on users and groups, although they are for compatibility still supported, actually now perform operations on groups, and this is explained in the documentation.

Lovely - pure, simple, utterly flexible, easy to reason about.

We now however find something also and unfortunately called "roles" in Redshift.

As has been stated, these are *not* Postgres roles. To my eye, what we see here is what I think we often see in Redshift, which is implementation constrained by a large, legacy code base : in other words, something bolted on the side, typically with a very large dose of misleading marketing.

In Redshift, we still have users and groups, and they remain separate, first-class types in the database, but now we have roles *as well*. Users are *not* roles, groups are not roles, the Postgres tables for roles have *not* turned up in the system tables; there is no `pg_roles`. There *is* now a `pg_role`, but it's different to the Postgres system table for roles, and there are a couple of extra, non-Postgres tables, to go along with it.

None of these new system tables are user accessible, which is great for AWS and their culture of secrecy, but bad for users, because it means we've stuck with whatever system tables (views, really) the devs put on top of these tables, and the devs are *not* good at system tables - and indeed, as we will see, they've messed this up in a couple of ways.

Roles in Redshift are a collection of privileges, just like a group, but roles (unlike groups) can be granted to roles and - and this is the biggie - there are a set of

new privileges, which can only be granted to roles; not to users, not to groups.

What this means in principle (but quite possibly not in practise, as we will see, due to implementation concerns) is that roles have superseded groups; everything you do with groups you can do with roles, and roles give you something new (the new Redshift privileges).

So rather than managing privileges by creating groups and granting privileges to groups, and then adding or removing users from groups, we now manage privilege by creating roles and granting privileges to roles, and then granting or revoking roles from users, and also potentially granting roles to roles, allowing a hierarchy of roles to be built up.

What's holding me back from roles is basically the question of whether or not I trust the implementation to be correct. I'm pretty confident groups, users and Postgres-style privileges work. I'm not confident about anything new from Redshift, because I've repeatedly seen over many years that testing is minimal or seemingly non-existent and this lack of confidence when we're looking at access controls which can then involve PII and legal obligations is a concern.

There are other issues.

The implementation of roles is not Postgres-compliant or backwards compatible (note here the Postgres implementation of roles *is* backwards compatible, and so existing Postgres tooling continued to work when roles were introduced) and so existing tooling, either native or from Postgres, which uses the `ACL` columns to understand group privileges, will have no knowledge of roles at all.

Regarding the use of sub-roles, I could be completely wrong, but I can't really see much mileage in this. The number of groups/roles should be kept as small as possible, to make them easy to reason about, and I think there usually is only a need for a very few groups or roles - there usually are only a few distinct sets of users which need different privileges - so I can't see a *need* for this extra organizational capability. Life I would say is usually simple enough that single-level groups (or single-level roles) is enough.

The new Redshift privileges are much more consequential, however - but now it's surprise time, or maybe not such a surprise; the way they are arranged and the way you use them is completely different to how the existing privileges are arranged.

What's really interesting here is what we can see of the implementation.

Back in about 2019, almost all system tables were converted from being tables to being views, with the underlying table no longer being accessible to users.

Part of the implementation of roles is to add code to *every single system table view.*

Here's the text for a system table `svv_roles` (I just happen to pick `svv_roles` - all of the system tables have had the same code added to them).

(Please note the code you see here is *nothing* like the code you get from Redshift, for this view; I have completely reformatted the code, moving it largely but not quite fully (some adaptations to deal with the exigencies of this particular code) to my own style, which makes it readable. I would note I've removed very large

numbers of unnecessary brackets, as well as unnecessary quoting of function names, unnecessary casts and so on, which which makes me think the author is an automated tool of some kind. Be aware also you cannot run the code here, as the system tables `pg_role` and `pg_identity` are accessible only to the AWS root user, `rdsdb`.)

```
select
  pg_role.rolid                     as role_id,
  pg_role.rolname::varchar(128)     as role_name,
  pg_identity.usename::varchar(128) as role_owner,
  pg_role.externalid::varchar(128)  as external_id
from
         pg_role
    join pg_identity on pg_identity.useid = pg_role.rolowner
where
      pg_role.rolname     !~~ '/%'
  and pg_identity.usename !~~ 'f346c9b8%'
  and
  (
    exists
    (
      select 1
      from   pg_identity
      where  pg_identity.useid = current_user_id and pg_identity.usesuper = true
    )
    or has_system_privilege( current_user, 'access system table' )
    or user_is_member_of(    current_user, pg_role.rolname       )
    or current_user_id = pg_role.rolowner
  );
```

Most the `where` clause is new code, and it's there to implement roles.

Let's walk through the `where` clause;

1. Show rows to the user where the role name does *not* begin with a forward slash (which if you try to use it in a role name, is an invalid character). This view is leader node only, so I think the use of `not like` where will *not* invoke `AQUA`, which is a good thing (high initial cost, then multiple queries needed to recoup that cost).
2. Show no roles owned by user `f346c9b8`.
3. If the user is a super user, show the row.
4. If the user is not a super user, but holds the Redshift-style privilege `access system table`, show the row.
5. If the user is not a super user, and does not hold the Redshift-style privilege `access system table`, but the user has the role granted to him, show the row.
6. If the user owns the role, show the row.

So… observations.

It's too much code and too much work - the actual code of the view is now outweighed by the boilerplate. In some views its worse - in `svv_role_grants` because of the way the view works the boilerplate code has to be present twice, and so you have six lines of real code and about fifty lines of boilerplate, with

two extra joins and four extra `select`s.

The Redshift-style privilege `access system table` is implemented in view SQL code; it has to be implemented correctly in every single view, and there are *lots* of system table views. This, on the face of it, seems crazy; security requires reliability - correct implementation, which in turn requires simple and compact implementation, which is easy to test. Approaches which are risky are inherently insecure. The code for security access should be in one place only, not at every possible entry-point. I am also concerned here about the poor reputation of Redshift for testing.

And now for a biggie; all of the system table views which now look like this have been made *leader node only*.

This is because the `has_system_privilege()` and `user_is_member_of()` functions are leader node only.

I think this is going to break a bunch of existing code out in the wild.

Finally, we can see that we see the Redshift-style privilege `access system table` is also conferring the powers of `syslog unrestricted`; not only can you now `select` from all views, but you also get to see all rows, not just your own.

Moving on, we see as expected where this is unlike the Postgres implementation of roles, role names are *not* showing in the `ACL` columns in system tables. The only way to know about roles is via some new, Redshift-specific system tables, such as `svv_roles`.

These new system tables, to my considerable surprise, do not show all rows to a user with `select` on the table and `syslog unrestricted` - they only show the rows owned by the user. This is not expected, and not consistent with prior behaviour in all other system tables.

These system tables will only show all rows if the user holds the new Redshift privilege, `access system table`.

I don't know if this means the old security model is now lapsed, or if it's an oversight, or a blunder. It's a problem, because the only way I can now grant access to the rows in these system tables is by granting access to the rows in *all* system tables. Previously, I could pick and choose exactly which system tables I gave access to.

Finally, with Postgres-style privileges, the `GRANT` command provides the stanza `WITH GRANT OPTION`. Normally, which is to say without `WITH GRANT OPTION`, a user holds a privilege and so he can then perform the action permitted by the privilege, and that's it; the user could not grant that privilege to *other* users, he could only use that privilege himself.

However, when a privilege is granted using `WITH GRANT OPTION`, the user then additionally is permitted to grant that privilege to other users.

Roles have something akin to this, which is called `WITH ADMIN OPTION`, which is present in the `GRANT` syntax when you're granting a role. The docs however have made a special effort to be especially opaque in this matter, providing and

only providing this one line of text to explain what option is for and how it is used;

> The WITH ADMIN OPTION clause provides the administration options for all the granted roles to all the grantees.

So, the way it works is that being able to further grant a Redshift privilege is *not* something which is granted to roles - it is granted to a user when the role is granted to the user.

When you grant a role to a user, it's *then* you indicate `WITH ADMIN OPTION`, and this means that user now has the privilege to grant that particular role to other users.

So it's not the capability to grant to other users a particular privilege - it's the capability to grant to other users a particular role.

That's a pretty powerful capability (and deserving far more than a single line of incomprehensible documentation, especially given that we're talking about the security model for a database), because roles I suspect are generally going posses at least a couple of Redshift-style privileges, and those privileges as we've seen are global, everywhere and always, and also because a role can contain many Postgres-style privilege, and the capability to grant those multiple Postgres-style privileges is also being conferred, although, of course, only *en bloc*, as the user can grant only the role, not the individual privileges which are granted to the role.

The Postgres-style privileges, where they are a single privilege on a single object, inherently minimize the capability being granted. The equivalent of roles with Postgres-style privileges would be that you grant a bunch of privileges to a group, and then grant a user the privilege to add users to that group - but this capability, to add users to groups, is available only to super users. There's no Redshift-style or Postgres-style privilege to allow granting it to ordinary users, which is a shame.

I would say that roles, and the Redshift-style privileges mechanism, are inherently going to tend to be significantly more consequential when they go wrong, than the Postgres-style privileges mechanism.

Postgres Privileges

So, there are almost all well known, and I've included them here for completeness. Only the `drop` privilege is new, something added to Redshift in about 2023.

Object	Privileges
Database	create, temporary
Function	execute
Language	usage
Procedure	execute
Schema	create, usage
Table	drop, insert, references, select, update
View	drop, select

Database

Privilege	Function
create	The `create` privilege allows the user to create schemas in the given database.
temporary	The `temporary` privilege allows the user to create temporary tables in the given database (which is to say, to issue. `create temp table`). This privilege is not needed to use CTEs.

Function

Privilege	Function
execute	The `execute` privilege allows the user to execute the function.

Language

Privilege	Function
usage	The `usage` privilege allows the user to create objects which use the given language. It is not required to execute objects in the given language.

Procedure

Privilege	Function
execute	The `execute` privilege allows the user to execute the given procedure (which includes functions).

Schema

Privilege	Function
create	The `create` privilege allows the user to create objects (which is to say, anything where a schema is a valid concept sense - a function, procedure, table or view - in the given schema.
usage	The `usage` privilege allows the user to perform actions on objects in the schema. Without it, holding privileges on objects in the schema is meaningless, as you the user will not be permitted to perform any actions on objects in the schema.

Usage is a per-schema toggle, basically; when absent, it blocks all actions on all objects in the schema, regardless of whatever privileges a user holds.

Table

Privilege	Function
drop	The `drop` privilege allows the user to drop the given table. This is new, introduced late 2022. It's not fully ramified - there's no drop privilege for say databases, functions, procedures; just tables (and views, where tables and views are treated as much the same, under the hood).
insert	The `insert` privilege allows the user to insert rows into the given table.
reference	The world's least used privilege, ever. The `reference` privilege allows the user to create a foreign key constraint in the given table, but not you must also hold this privilege on the foreign table, too.
select	The `select` privilege allows the user to select rows from the given table.
update	The `update` privilege allows the user to update rows in the given table.

View

Privilege	Function
drop	The `drop` privilege allows the user to drop the given view. This is new, introduced late 2022. It's not fully ramified - there's no drop privilege for say databases, functions, procedures; just views (and tables, where tables and views are treated as much the same, under the hood).
select	The `select` privilege allows the user to select rows from the given view.

Remember that when you select rows from a view, the view will access the tables it uses as if it were the *owner* of the view. If the owner has the privilege to select from the tables, then you're fine - and this is also how you use a view to provide access to tables, without actually granting the privilege to select on those tables.

If the owner does not have the necessarily privileges, then the user trying to select from the view will be presented with a missing-privileges error (on behalf, as it were, of the owner of the view).

Redshift Privileges

Let's now examine the new, Redshift-style privileges.

What are they, and what do they do, and how are they organized?

To begin with, we turn to the official documentation, which we find here.

I originally wrote here at this point a lot here about the shortcomings of the docs, but it's just not worth the time to read.

Suffice to the say the docs are almost completely without value. They are superficial, take a page at a time to convey a single fact (a bit like the v2 Redshift console!), and lack almost all of what would have be considered absolutely basic and essential information, such as a list of the Redshift-style privileges and what they actually do.

So, let's begin by enumerating the new privileges.

This is not straightforward. There are the official docs, which has a page presumably intended to list all privileges (but it doesn't, and also the main information on that page is what privileges you need to hold to *grant* each privilege, not what each privilege *actually does*), then there's the `GRANT` command, which has in its syntax a slightly different list of privileges, and then there's what we find by examining the system table `svv_system_privileges`, which lists all granted Redshift-style privileges, and what you get from that is quite different to both of the above.

So here's what we find in each of these sources of information.

docs	`GRANT` doc page	`svv_system_privileges`
-	-	access system table
alter datashare	alter datashare	alter datashare
alter default privileges	alter default privileges	alter default privileges
-	-	alter materialized view
		row level security
alter table	alter table	alter table
-	-	alter table enable row
		level security
alter user	alter user	alter user
analyze	analyze	analyze
-	-	attach rls policy

docs	`GRANT` doc page	`svv_system_privileges`
cancel	cancel	cancel
create datashare	create datashare	create datashare
create library	create library	create library
create model	create model	system create model
create or replace external function	create or replace external function	create or replace external function
create or replace function	create or replace function	create or replace function
create or replace procedure	create or replace procedure	create or replace stored procedures
create or replace view	create or replace view	create or replace view
-	-	create rls policy
create role	create role	create role
create schema	create schema	create schema
create table	create table	create table
create user	create user	create user
-	-	detach rls policy
drop datashare	drop datashare	drop datashare
drop function	drop function	drop function
drop library	drop library	drop library
drop model	drop model	drop model
drop procedure	drop procedure	drop procedure
-	-	drop rls policy
drop role	drop role	drop role
drop schema	drop schema	drop schema
drop table	drop table	drop table
drop user	drop user	drop user
drop view	drop view	drop view
explain rls	-	explain rls
-	-	grant role
ignore rls	-	ignore rls
truncate table	truncate table	truncate table
vacuum	vacuum	vacuum

Note;

1. `create or replace procedure` is called `create or replace stored procedures` in `svv_system_privileges`
2. `system model` is called `system create model` in `svv_system_privileges`

Of the privileges listed in `svv_system_privileges`, you cannot grant the following;

1. alter materialized view row level security
2. alter table enable row level security
3. attach rls policy
4. create rls policy
5. detach rls policy
6. drop rls policy

These privileges are held by one of the five built-in, pre-existing, roles, `sys:secadmin`. The docs describe this role thus;

> This role has the permissions to create users, alter users, drop users, create roles, drop roles, and grant roles. This role can have access to user tables only when the permission is explicitly granted to the role.

In fact this role is the only way to obtain these particular Redshift privileges, and this critical fact is not mentioned.

Moving on, I think it's reasonable to conclude there isn't any test code which is granting every Redshift-style privilege and then checking it has been granted, because that code would notice that two privileges have non-matching names in the `svv_system_privileges` system table - and note here that roles came out in April 2022, which at the time of writing, was nine months prior.

The next observation I would say is that there are a lot of privileges.

AWS seem to be taking their usual route of providing very granular permissions - think `IAM`.

That's good and bad. The good is you can exactly specify what you want to do, and I can see that the capability to so exactly specify what you want, is necessary given the vast number of different use cases out there; on the other hand, it can become overwhelming.

All things considered, given the need to support an almost infinite number of uses cases, I think AWS in this are doing the right thing, but they've also going against themselves, by making the privileges global - everywhere and always. They would be much finer grained if they could be granted for single objects.

(Redshift-style privileges cannot be granted to users, but you can make one role per user - with the same names - and grant to that role, emulating the capability to grant to single users. Clunky, but entirely viable.)

Now we've enumerated the privileges, let's look at what they do; but to do this properly, where each privilege is on the face of it a black box, I would have to implement a test suite which tests every single aspect of Redshift behaviour, then grant one privilege, and see what changes. That would be thorough, but it's too big an ask. Instead, I've taken each privilege in turn and assumed its name reflects what it does, and then manually experimented with a cluster to find answers to the questions which came to mind for that particular privilege.

For some privileges (row-level security, libraries, models, etc), I've yet to investigate or even use that functionality in Redshift, and so to be able to think of questions for those would require investigating each area, which again is too big an ask for this right now.

Privilege	Function
access system table	Provides `select` access to all publicly accessible tables and views in `pg_catalog` and `information_schema`. This privilege also confers `syslog unrestricted`; the user will always see all rows, not just rows for objects the user owns. (Note however both this privilege and `syslog unrestricted` do not work for system function which generate rows, which are used in an increasing number of system table views - to see such rows, the only way is to be a super user.)
alter datashare	Not investigated.
alter default privileges	Allows the user to modify default privileges for all users (including super users).
alter materialized view row level security	Not investigated.
alter table	Allows the `alter table` command to be issued on any table (any normal Redshift table, that is). Note this include the capability to change the owner, so really this privilege gives complete control over all tables (and views, as tables and views are seen as the same, under the hood).
alter table enable row level security	Not investigated.
alter user	Allows the `alter user` command to be issued on any user. A user with this privilege can make himself super user, so really this privilege is the same as being super user.
analyze	Allows the user to issue `analyze` on any table (any normal Redshift table, that is).
attach rls policy	Not investigated.
cancel	Allows the user to issue `cancel` on any process, except I suspect those owned by `rsdsb` - but this is difficult to test, as such queries are fleeting.
create datashare	Not investigated.
create library	Not investigated.
create model / system create model	Not investigated.
create or replace external function	Not investigated.

Privilege	Function
create or replace function	Allows the user to create, or replace, any function in any schema in any database.
create or replace procedure / create or replace stored procedures	Allows the user to create, or replace, any procedure in any schema in any database. If you hold this privilege, and you want to do my PL/pgSQL for me, that'd be just fine :-)
create or replace view	Allows the user to create, or replace, any normal or late-binding view in any schema in any database. Does not work for materialized views, as they cannot be replaced (only dropped and then re-created). Note that when replacing a view, the column names and types must be unchanged, although I think there's some flexibility in types (varchar lengths can change, for example), but that's beyond scope here.
create rls policy	Not investigated.
create role	Allows the user to create roles.
create schema	Allows the user to create schemas, in any database.
create table	Allows the user to create normal Redshift tables in any schema, in any database. This includes temporary tables.
create user	Allows the user to create users, which means being able to create a super user, and then log in as that super user.
detach rls policy	Not investigated.
drop datashare	Not investigated.
drop function	Allows the user to drop any function, in any schema, in any database, except those owned by `rdsdb`.
drop library	Not investigated.
drop model	Not investigated.
drop procedure	Allows the user to drop any procedure, in any schema, in any database, except those owned by `rdsdb`, but Redshift ships with no procedures, so you'd actually have to change the owner to `rdsdb` and that's actually not allowed :-)
drop rls policy	Not investigated.

Privilege	Function
drop role	Allows the user to drop roles. To drop a role, it must be revoked from all users, and have all privileges removed.
drop schema	Allows the user to drop any schema, in any database, except those owned by `rdsdb`.
drop table	Allows the user to drop any table, in any schema, in any database, except those own by `rdsdb`.
drop user	Allows the user to drop any user, including super users. As is normal though to drop a user, the user must own no objects or privileges, and so typically for this privilege to be meaningful, the holder must be able to change ownerships, and/or drop objects and privileges. There is no Redshift privilege which allows a user to change object ownerships; you must still be the object owner, or super user, to do this.
drop view	Allows the user to drop normal views and late-binding views, in any schema, in any database, but not materialized views (for this you need to use `drop materialized view`, and there is no Redshift-style privilege for this. With normal views, the usual dependency rules apply, so a view cannot be dropped if other objects depend upon it, unless the `cascade` option is used.
explain rls	Not investigated.
grant role	Allows the user to grant any role, to any user. This includes granting the built-in `sys:superuser`, which holds every Redshift-style privilege, including `alter user`, and by this the user can then elevate themselves to super user.
ignore rls	Not investigated.
truncate table	Allows the user to issue `truncate table` on any table (any normal Redshift table, that is), in any schema, in any database.

vacuum	Allows the user to issue `vacuum` on any table (any normal Redshift table, that is), in any schema, in any database.

So, the risky privileges are;

Privilege	
alter table	Allows users to take ownership of all tables and all normal views and late-binding views (but not materialized views).
alter user	User can make themselves super user.
create user	User can create a super user, then log in as that super user.
grant role	User can grant themselves the built-in role `sys:superuser`, which gives `alter user` and `create user`.

Note with `alter table`, even if the user inspects `pg_class` to find the names of the underlying table/view pair which are a materialized view, the user still cannot take ownership as they are both owned by `rdsdb`.

Default Privileges

When granting Postgres-style privileges, a privilege is granted at the moment it is granted, on the specified object, to the specified user; a grant never in any way applied to objects which do not yet exist.

The reason I say this is that in the `GRANT` syntax there is the formulation where you can grant privileges on all tables in a schema, like so;

```
grant select on all tables in schema dining_room to bob_the_skutter;
```

This in my experience is often misunderstood to mean "grant this privilege on all tables which currently exist in this schema, *and* all tables which will in the future be created in this schema".

In fact, all it means is "grant this privilege on all tables which currently exist in this schema".

The key is to realize this syntax is helper syntax only. It saves you having yourself to enumerate all the tables in a schema and issue the grant command on each table - all it does is enumerate the existing tables in the schema, and issue the grant on all of them. Future tables are brand new objects, wholly unaffected by earlier grants.

However, it would often be rather nice if when an object is created privileges of some kind upon it were automatically granted.

For example, we might have a group of BI users, and we will always want them to have access to every table in the schema `bi_aggregate_tables`.

Rather than having to remember when making a new table in that schema to issue the necessary grants to the BI user group, there is in fact a mechanism to do this for us - to automatically grant privileges when an object is created.

This mechanism is known as *default privileges*.

Default privileges are owned by users. Each user can have none, or many, default privileges.

A default privilege specifies an object type (function, procedure, or table (which includes views)), a single privilege (naturally, valid for the type of object), and a single recipient for that privilege (a user, a group, or the group-like object public); and when the user who owns the default privilege creates an object of that type, the given privilege is automatically granted to the recipient.

(There can be multiple default privileges for the same type of object, so creating a table might say grant `select` to a number of groups, each group requiring one default privilege for its grant, but inherently each default privilege is unique - to issue the same default privilege twice is to specify the exact same behaviour, for the exact same object, as already exists; it simply replaces the existing identical default privilege with a new, identical default privilege.)

This way when a user creates, say, a table, the privilege to `select` from it will automatically be granted to say a couple of different groups (this needing one default privilege per group, since each default privilege specifies a single privilege and a single recipient).

Finally, note that a default privilege can also have a specified a single schema, and when this is done, the default privilege operates only for objects created in that schema; and that a default privilege can have specified a single user, which is the user to own the default privilege. If the user is not specified, the current user owns the privilege - rather than *all* users, which can be a natural misinterpretation of the syntax.

Default privileges are central to organizing privileges, but as a mechanism it seems pretty unknown. I've seen a number of systems where the admin have built a manual system which issues `grant [priv] on all tables in schema [schema] to group [group]`, which they trigger when users complain about not being able to access tables.

With regard to Postgres and Redshift privileges, the situation is simple. Default privileges can issue and only issue Postgres-style privileges. This is expected, as default privileges specify an object, whereas Redshift-style privileges are global; Redshift-style privileges have no concept of an object, but are valid on all objects, always.

Conclusions

I think these new Redshift-style privileges would have been much more useful, and safer, if they have been as with Postgres-style privileges, on a per-user, and where applicable, a per-object basis. You can emulate per-user by making a role per user, which is clunky but viable, but per-object is not possible.

I may be wrong, but I think we've got what we've got, which is to say *global* privileges, because Postgres-style although better could technically not be done.

Looking at the implementation of roles, and the new Redshift privileges, they have materially complicated Redshift and its use. We now have users, group *and* roles, Postgres-style privileges *and* Redshift-style privileges, the bulky new SQL in the system tables views is awful, and and the docs, which always are very poor, are for roles *particularly* hopeless. Given how much power the Redshift privileges confer, this is a cause for concern.

I'm rather of the view the benefits of roles and the new privileges are not worth their cost in maintainability and complexity to Redshift as a whole, particularly so because I need strong confidence in new functionality relating to security.

There are however a couple of the new Redshift-style privileges which are particularly useful, and rather harmless; I am thinking of `analyze` and `vacuum`. Normally both can only be issued by the owner of a table, or a super user, it is very convenient for an ETL to issue these commands generally, and there is no PII risk.

There is also `truncate table`, which again is very useful for ETL, as normally only an owner or super user can issue this, but this is definitely not harmless - but, still, no PII risk. You could reasonably assign this to an ETL user.

What I often see in Redshift systems is that all objects are owned by an ETL user, with privileges granted by the ETL user to groups, to allow normal users access. It's a lot more natural for users to own the objects as appropriate to user's use, with the ETL system having the capability to perform operations anyway.

A couple of the new Redshift-style privileges (`alter user`, `create user`, `grant role`) are in fact properly tantamount to granting super user, as the holder can use them to elevate themselves to super user.

I note one or two omissions in the set of Redshift-style privileges; there are privileges relating to views, but not materialized views. There is also no privilege to take ownership of an object, but that may be by design, as it in fact conveys

complete power over all objects (but then so do the three privileges which allow elevation to super user).

All in all, my concerns about the reliability of implementation undercut the usefulness of the privileges. I'd be happy using a couple of the privileges with an ETL user - that would be very handy - but that's about it, because of the question of reliability.

Credits

1. Michael Bennett.

   For wisdom regarding default privileges, that it's possible to think if the user is omitted, the privilege is applying to everyone, when in fact it's applying to the current user only.

Revision History

v1

- Initial release.

v2

- Rewrote abstract.

v3

- Added text to "Default Privileges" to mention if the user is not specified, the default privilege is owned by the current user, not *all* users.
- Added the "Credits" page.

v4

- Changed to Redshift Research Project (AWS have a copyright on "Amazon Redshift").

v5

- Some minor editing in the section "Roles".

v6

- General review and editing work to improve the prose. No additional investigative work.

v7

- Substantial re-working. Document is much improved. No additional investigative work.
- Added "About the Author". made site name in title a link, and made each chapter start a new page.

v8

- Fixed a really serious typo in the first chapter, where I had “group” when I meant to have “role”, and a bit more rewriting of the first chapter.

v9

- One or two minor paragraph rewrites, but fixed a bad typographic error in “Group-like Object Public”, where a bit of text from a prior version of a sentence had remained in the new sentence (and making it a mess).

About the Author

I am a C programmer - kernel development, high performance computing, networking, data structures and so on.

I read the C. J. Date book, the classic text on relational database theory, and having learned the principles, wrote a relational database from scratch in C, which purely by chance set me up quite nicely for what came next, moving into data engineering in late 2011, when I joined as the back-end engineer two friends in their startup.

In that startup, I began using Redshift the day it came out, in 2012 (we had been trying to get into the beta programme).

We were early, heavy users for a year and a half, and I ending up having monthly one-to-one meetings with one of the Redshift team managers, where one or two features which are in Redshift today originate from suggestions made in those meetings, such as the distribution style `ALL`.

Once that was done, after a couple of years of non-Redshift data engineering work, I returned to Redshift work, and then in about mid-2018 contracted with a publisher to write a book about Redshift.

The book was largely written but it became apparent I wanted to do a lot of things which couldn't be done with a book - republish on every new Redshift release, for example - and so in the end I stepped back from the contract and developed the web-site, where I publish investigation into, and ongoing monitoring of, Redshift.

So for many years now I've been investigating Redshift sub-systems full-time, one by one, and this site and these investigations are as far as I know the and the only source of this kind of information about Redshift.

Redshift Cluster Cost Reduction Service

I provide consultancy services for Redshift - advice, design, training, getting failing systems back on their feet pronto, the usual gamut - but in particular offer a Redshift cluster cost reduction service, where the fee is and only is one month of the savings made.

Broadly speaking, to give guidance, savings are expected fall into one of two categories; either something like 20%, or something like 80%. The former is

for systems where the business use case is such that Redshift cannot be operated correctly, and this outcome requires no fundamental re-engineering work, the latter is for systems where Redshift can be operated correctly, and usually requires fundamental re-engineering work (which you may or may not wish to engage in, despite the cost savings, in which case we're back to the 20%).

Details and contact information are on the web-site.