Users, Groups, Roles and Privileges

Max Ganz II @ Redshift Research Project

5th March 2023

Abstract

In Postgres, roles replaced users and groups (both become roles). Roles in Redshift do not replace users or groups, but exist alongside them, as a third, separate, first-class type. Roles, like groups, can be granted privileges, but roles, unlike groups, can be granted to roles. Granting a role is in effect the same as adding a user to a group. 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-style privileges). These new privileges are unlike the existing privileges (which are per-user, per-object) as they are global; they apply to everything, everywhere, always, in all databases. They are essentially fragments of the super user. Finally, note the documentation for roles is particularly egregious, and 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

This white paper assumes you are a passably experienced Redshift admin or user, and so you know what users are, what groups are, and what privileges are.

The Privileges Mechanism

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

Each type of object by being different in its nature has a particular set of privileges associated with it; for example, you can create temp tables in a database, but not in a function, so the privilege to create temp tables exists for databases but not for functions.

Privileges can be granted to users, groups and roles, but groups and roles are (as will be described) simply mechanisms to help organize the granting of privileges to users; groups and roles are entirely passive, for it is only users who actually perform actions, and so it is only users who are permitted or denied their actions according to the privileges they hold.

The privilege mechanism is however in one particular situation completely bypassed.

There is in Redshift (and Postgres) the concept of *ownership* for objects. 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, and the owner of an object can do anything they like with that object, always, because by being the owner, they in fact completely bypass the privileges mechanism.

(Objects can have their ownership changed, but they begin owned by the user who created them.)

The owner of an object can revoke from themselves every single privilege that exists upon an object, and still then be able to perform every possible action with that object.

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 since you'd need two billion users to overflow, you're likely to have a unique ID per user :-)

From the point of view of a human using Redshift, users are uniquely identified by a name, an arbitrary string, which is up to 127 bytes of `UTF-8`. All users must at any given time have a unique name, but names which were used but have since been deleted, can be re-used.

Note however there are some places in the system tables where users are referred to by name only - their user ID is not provided - and in fact it's not always the *full* user name which is given. This, when it happens, it is a blunder, because it is not possible then to guarantee you can uniquely identify the user. So, for example, `stv_sessions` indicates the user by the first 50 bytes only of the user name. To my knowledge, 50 bytes is the longest safe user name.

(This issues crops up with particular strength and pervasiveness for database names; to my knowledge the longest safe database name is only 32 bytes, rather than the 127 it should be).

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, etc. For the user to be deleted there can be *no* traces of that user in the database *at all* and the `drop user` command performs absolutely none of this work - it's all down to you. Ridding 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, 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 you need to enumerate all their privileges - something which until very recently Redshift could not do, which in fact made the situation problematic; you needed to drop a user, but the user holds privileges, but you can't find out what those privileges are.

As such, what I often saw happening is that users are *not* deleted, but rather have their account modified so they can no longer log in (and their password changed for good measure).

There are three *types* of user; normal users, super users, and the root user, which has the name `rdsdb`, the root user being owned and operated by AWS.

The root user is God, and can do anything - you might be thinking the super users can do anything, but this is in fact not the case; superusers 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`, and so you cannot touch them.

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, it's not possible to tell if a table has the `auto` sort-type except by going through `svv_table_info`, which is a big, costly view (9.5kb of text), 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.

Super users are normal users in every respect except one; they *always* completely bypass the privileges mechanism. Super users can perform any operation, on anything, always - except for objects owned by `rdsdb`.

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 superusers; 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 since you'd need two billion users to overflow, you're likely to have a unique ID per group.

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 of `UTF-8`, and all groups must at any given time have a unique name, but names which were used but have since been deleted, 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 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.

Groups cannot own objects; only users own objects.

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

Now, 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.

You can grant privileges to public, and by doing so, you grant those privileges to every user in the cluster.

When you examine 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, here the "who" can be the name `public`, and this is the and the only place this name is found.

Now, Redshift (and Postgres) ship with a large number of functions, which are part of SQL, two examples of which are as `min()` and `max()` (but there are thousands more), the type conversion functions (which we normally use via the `::` operator), these days all of the PostGIS functionality Redshift implements, and so on.

All of these functions are available for everyone to use by execute privileges being granted to public.

I have heard of systems where the admin, I believe for security purposes, want to eliminate public. You cannot do this - it is not a group, but an inherent part of Redshift/Postgres - the nearest you can get is to revoke all privileges from public, which makes the automatic membership of public wholly without effect.

Actually doing this is problematic.

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.

Secondly, if you do this, and revoke all privileges granted to public, you will then need to grant the thousands of privileges to built-in functions to a group of your own, to which you add users, so they can actually write SQL, because otherwise users will not be able to call any of the thousands of functions which are built in to SQL.

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.

Roles

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

In Postgres, it used to be (up to version 8.1) 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.

After Postgres 8.1, 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. ‘

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

Now we come to Redshift, and to my eye, what we see here is what we often see, which is an implementation constrained by a large, legacy code base : in other words, something bolted on the side.

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 table, 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 can be granted to roles (unlike groups) and there are a set of new privileges, Redshift-specific, which can only be granted to roles - not to users, not to groups.

What this means, in principle if potentially not in practise, is that roles have superseded groups; everything you do with groups you can do with roles, and roles give you something new. Groups are now obsolete.

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.

What we get that's new is that we can build roles up out of sub-roles (by granting roles to roles), and we get to issue the new Redshift-specific privileges, because they can only be granted to roles. We can still issue all existing Postgres-style privileges to roles, so we lose nothing.

What's holding me back from this 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 - access control which can then involve PII and legal obligations - can be highly sensitive. This is not a case of say access to tables in Postgres being unreliable or having odd flaws; this is a case of PII leaking and the company being legally liable. A higher bar must be met.

There are one or two 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.

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

Postgres privileges are arranged on the basis of a privilege per action, per object, per user; user Frodo has privilege usage on schema ring.

Redshift privileges, by contrast, are arranged *globally*; when a user is granted a Redshift privilege, that privilege is always active, on all objects, in all databases, always - regardless of whatever Postgres privilege are or are not granted.

So here we see user Gandalf has privilege create table - simple as that; there's no object. Gandalf can now create tables, everywhere, always. It's good to be Gandalf!

In short, Redshift privileges are like fragments of being a super user. They give the user the capability to perform some action, but everywhere, and always.

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

Here's the text for the system table (it's a view really) `svv_roles`.

(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 you cannot run this code, 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
  );
```

There's a lot to say here, but I'll start by examining the `where` clause, which is controlling whether or not rows are shown to the user.

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 - the actual code of the view is now outweighed by the boilerplate. In some views its worse - in `svv_role_grants`) the boilerplate code is 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 row 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` syntax 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. However, when a privilege is granted using `WITH GRANT OPTION`, the user then additionally is permitted to then, themselves, 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 make a special effort in this matter, with this one line of text;

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

Feynman once was asked to review a whole bunch of physics textbooks. It drove him crazy - he said, none of them actually *explain* anything; that you could replace the words being used with nonsense words and the power of what was written to convey meaning would not be harmed by it.

In any event, Redshift-style privileges are granted to roles only - not to users, not to groups - so the idea of granting a Redshift-style privilege to a role but also indicating the role can itself then grant that privilege to others, makes no sense. Roles can't grant.

So the way it works is that 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.

That's a pretty powerful capability, 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 for it.

All in all, 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.

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.

Privilege	Function
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 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 I'm afraid almost completely without value. They are superficial, take a page at a time to convey a single fact, and lack almost all of what would have be considered 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 (which tells you what privileges you need to *grant* each privilege, not what each privilege *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 that's quite different.

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
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	create or replace	create or replace
external function	external function	external function
create or replace	create or replace	create or replace
function	function	function
create or replace	create or replace	create or replace stored
procedure	procedure	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 roles, `sys:secadmin`. The docs incorrectly 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.

The docs in general have issues, and what's seen here is in line with previous and existing issues. You cannot rely the docs; read them, but keep in mind they may well be wrong, and in any way you can think of =-) It is not safe to simply fully accept what's written as correct.

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 found answers to the questions which come 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.
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.

Privilege	Function
create library	Not investigated.
create model / system create model	Not investigated.
create or replace external function	Not investigated.
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.

Privilege	Function
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.
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.

Privilege	Function
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".

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, as ever, are completely hopeless.

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.

I must say here that the official docs for roles are really, *really* no good; bad enough they actually merit special mention.

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").