Role-based Access Control (RBAC) with Secondary Roles

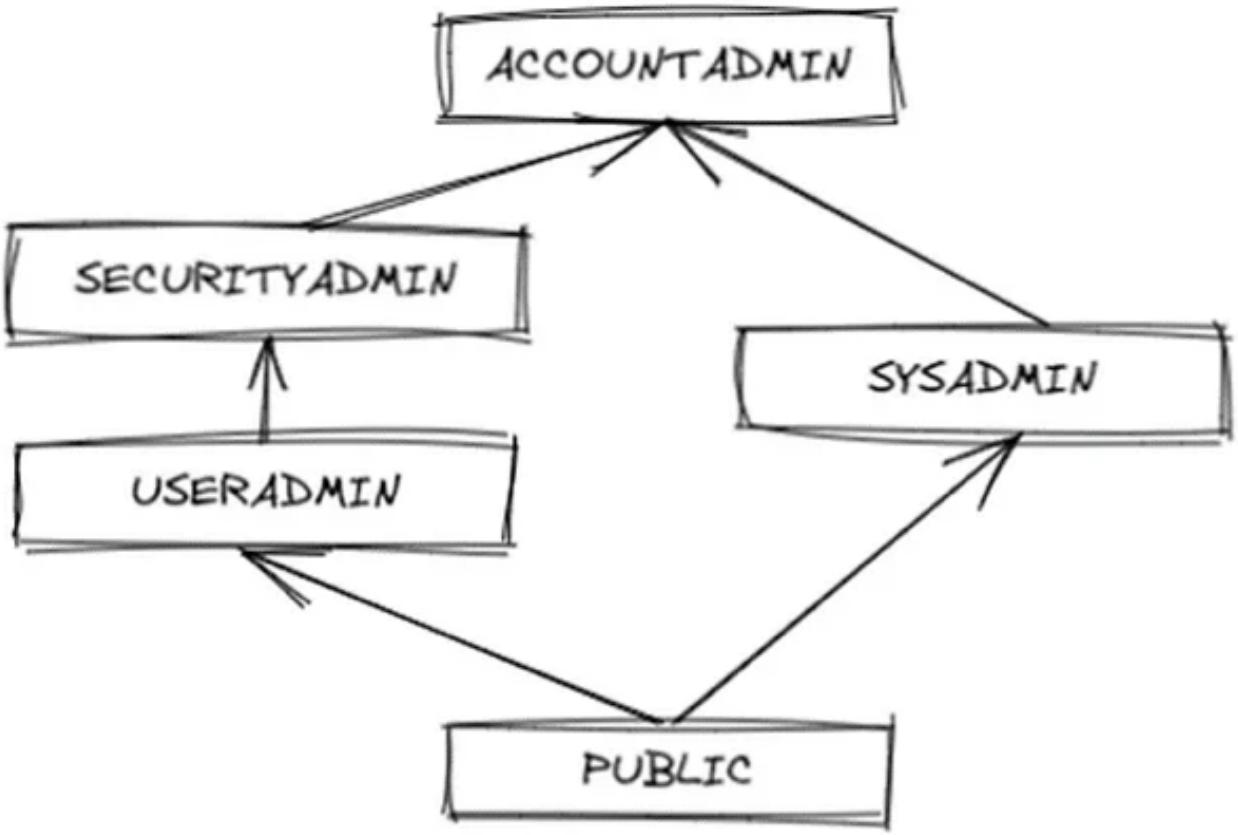
Maja Ferle · Following

Published in Snowflake Builders Blog: Data Engineers, App Developers, AI/ML, & Data Scien… ·
10 min read · Oct 10, 2022

Photo by Florian Rieder on Unsplash

Setting up Role-based Access Control (RBAC) in Snowflake requires careful planning and design of the roles, environments, and database objects that will be involved. Reusable scripts or stored procedures can be written to create the objects and roles in each environment. These scripts allow centralized maintenance of the RBAC implementation and when done correctly, RBAC is a powerful approach to manage user access and privileges in Snowflake.

Before delving into RBAC design, let's briefly review the Snowflake system roles and their purpose.



Snowflake system roles

ACCOUNTADMIN — top-level role that should be used for account level configuration and maintenance only.

SECURITYADMIN — powerful role with the MANAGE GRANTS privilege that can modify or revoke any grant in the account, as well as create, monitor, and manage users and roles. This role is typically not granted any custom roles and as a result it doesn't have access to database tables.

USERADMIN — this role usually creates users and roles as well as manages users and roles that it owns.

SYSADMIN — primary role that creates objects such as warehouses, databases, and other objects in the account as well as grants and revokes privileges on objects. Typically, this role can access all data and can assume all roles.

PUBLIC — this role is automatically granted to every user and role. Any objects owned by the PUBLIC role are accessible to every user and role.

In Snowflake, there is a clear separation between the roles that manage users and grants (SECURITYADMIN, USERADMIN) and the role that manages objects (SYSADMIN). We will want the same separation when creating our RBAC environments. This will allow us to grant the environment system administration role to one or more members of the development team so that they will be able to create their own database objects. By keeping the role that manages users and grants separate from the system administration role, another administrator can be assigned this role to manage who has access to the environment and which roles they may use.

When creating roles that will own objects, Snowflake recommends creating a hierarchy of custom roles, with the top-most custom role assigned to the system role SYSADMIN. This role structure allows system administrators to manage all objects in the account.

To design RBAC for an environment, consider the following activities:

- Define a set of functional roles that will be granted to users according to how they will be using the environment. For example, functional roles may be *data analyst*, *data engineer*, *data scientist*, and so on. Some of these roles, such as *data analyst*, may be too generic. You might further refine functional roles according to the business functions within the organization, such as *finance analyst* or *marketing analyst* or *HR analyst*. These roles often align to users based on their job functions.
- Define a set of access roles with a common set of privileges, for example we usually need a role that has ownership of objects in each schema, another role with read and write access, and a role with read-only access.
- Access roles should be granted to functional roles according to the required privileges of each functional role. Functional roles are granted to users. To keep it simple, do not grant access roles to other access roles and if possible, try not to grant functional roles to other functional roles.
- Use managed access schemas to prevent object owners from granting access to other roles at their discretion and to centralize grant management. Define future grants on all object types in the schema, meaning that all newly created objects in the schema will receive the same grants to access roles as the current objects.
- Define environment administration roles which will correspond to the SYSADMIN and USERADMIN system roles. We want a role that will own the objects in the environment and will act as the environment administrator for creating and maintaining objects and granting access on these objects to the access roles. We want another role that will

create the users and roles within the environment.

As an example, let's set up an environment named ENV. Depending on the size of the organization and the software development lifecycle, environments may be defined separately for development, testing, UAT, production, and so on. Additionally, there may be separate environments per business units or per subject areas. It's always a good idea to start with a set of abbreviations that will be used to indicate the environment name and then use these abbreviations when creating objects and roles.

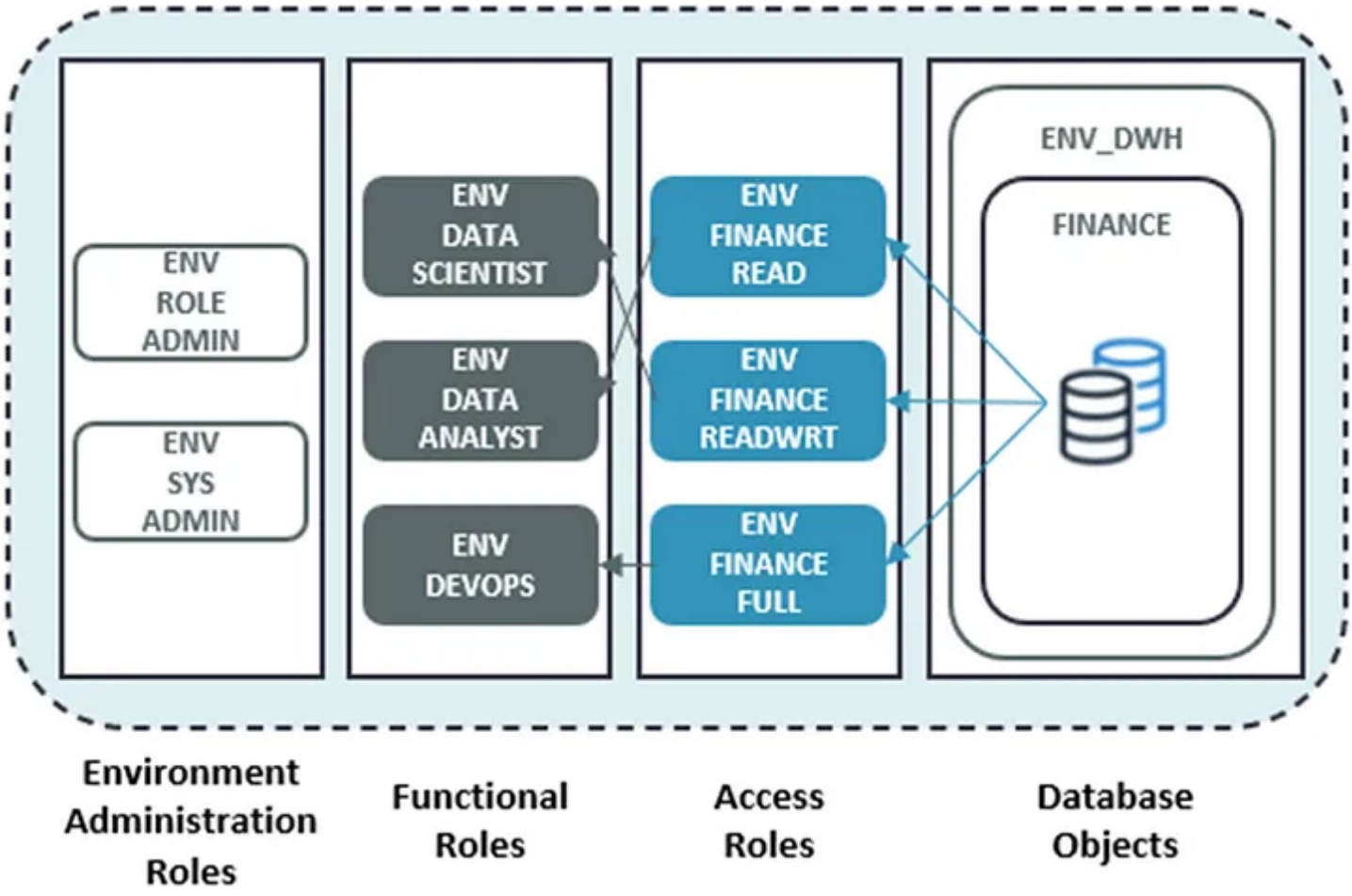
We will define 3 functional roles: *data analyst*, *data scientist*, and *devops*. These roles will be named ENV_DATA_ANALYST, ENV_DATA_SCIENTIST, and ENV_DEVOPS respectively.

We will define 3 access roles for each of the schemas in the environment: read, read/write and full. These roles will use suffixes _READ, _READWRT and _FULL respectively.

The two environment administrators will be named ENV_ROLE_ADMIN and ENV_SYS_ADMIN.

We will also create a database named ENV_DWH and a schema named FINANCE.

All of these roles and objects are represented in the following diagram:



Environment roles and objects

To set up the scripts that will create all of these roles and objects, it takes a bit of patience and careful planning, especially since we are using two environment administration roles and we must switch between them to get everything set up and working as expected.

Our first step will be to create the environment management roles using the USERADMIN role:

```
use role USERADMIN;
```

```
-- create the environment role administrator
create role ENV_ROLE_ADMIN;

-- grant privilege to enable creating roles
grant create role on account to role ENV_ROLE_ADMIN;

-- create the environment system administrator
create role ENV_SYS_ADMIN;

-- grant the environment system administrator role to SYSADMIN
grant role ENV_SYS_ADMIN to role SYSADMIN;
```

Then we will grant these newly created roles to ourselves (the current user), so that we will be able to use them later, still using the USERADMIN role:

```
set my_current_user = current_user();
grant role ENV_ROLE_ADMIN to user identifier($my_current_user);
grant role ENV_SYS_ADMIN to user identifier($my_current_user);
```

To allow the environment SYSADMIN to create objects in the account, we must use the SYSADMIN role to grant the CREATE DATABASE privilege:

```
use role SYSADMIN;

-- grant privilege to enable creating databases
grant create database on account to role ENV_SYS_ADMIN;
```

Now we will create a database and a schema with managed access using the ENV_SYS_ADMIN role:

```
use role ENV_SYS_ADMIN;

create database ENV_DWH;
create schema FINANCE with managed access;
```

After we have created the database and one or more schemas, we can create access roles for each of the schemas using the ENV_ROLE_ADMIN role:

```
use role ENV_ROLE_ADMIN;

-- create access roles
create or replace role ENV_FINANCE_READ;
create or replace role ENV_FINANCE_READWRT;
create or replace role ENV_FINANCE_FULL;

grant role ENV_FINANCE_READ to role ENV_SYS_ADMIN;
grant role ENV_FINANCE_READWRT to role ENV_SYS_ADMIN;
grant role ENV_FINANCE_FULL to role ENV_SYS_ADMIN;
```

We have also granted the newly created access roles to the environment system administrator role ENV_SYS_ADMIN. Due to privilege inheritance and role hierarchy, since we have granted the ENV_SYS_ADMIN role to SYSADMIN earlier, the SYSADMIN role will inherit these roles automatically.

Now we will use the ENV_SYS_ADMIN role again to grant usage privileges on the database ENV_DWH and schema FINANCE to the newly created roles:

```
use role ENV_SYS_ADMIN;

-- grant database usage
grant usage on database ENV_DWH to role ENV_FINANCE_READ;
grant usage on database ENV_DWH to role ENV_FINANCE_READWRT;
grant usage on database ENV_DWH to role ENV_FINANCE_FULL;

-- grant schema usage
grant usage on schema FINANCE to role ENV_FINANCE_READ;
grant usage on schema FINANCE to role ENV_FINANCE_READWRT;
grant all privileges on schema FINANCE to role ENV_FINANCE_FULL;
```

Since we have a newly created database and schema, we expect that there are no objects created yet and we don't have to grant any access on the current objects. We will grant only future access for objects that will be created in the schema in the future:

```
-- grant read access on future objects
grant select on future tables in schema FINANCE
to role ENV_FINANCE_READ;
grant select on future external tables in schema FINANCE
to role ENV_FINANCE_READ;
grant select on future views in schema FINANCE
to role ENV_FINANCE_READ;
grant usage, read on future stages in schema FINANCE
to role ENV_FINANCE_READ;
grant usage on future file formats in schema FINANCE
to role ENV_FINANCE_READ;
grant select on future streams in schema FINANCE
to role ENV_FINANCE_READ;
grant usage on future procedures in schema FINANCE
to role ENV_FINANCE_READ;

-- grant read/write access on future objects
grant select, insert, update, delete, references on future tables in
schema FINANCE to role ENV_FINANCE_READWRT;
grant select on future external tables in schema FINANCE
to role ENV_FINANCE_READWRT;
grant select on future views in schema FINANCE
```

```
to role ENV_FINANCE_READWRT;
grant usage, read, write on future stages in schema FINANCE
to role ENV_FINANCE_READWRT;
grant usage on future file formats in schema FINANCE
to role ENV_FINANCE_READWRT;
grant select on future streams in schema FINANCE
to role ENV_FINANCE_READWRT;
grant usage on future procedures in schema FINANCE
to role ENV_FINANCE_READWRT;
grant usage on future functions in schema FINANCE
to role ENV_FINANCE_READWRT;
grant usage on future sequences in schema FINANCE
to role ENV_FINANCE_READWRT;

-- grant full access on future objects
grant ownership on future tables in schema FINANCE
to role ENV_FINANCE_FULL;
grant ownership on future external tables in schema FINANCE
to role ENV_FINANCE_FULL;
grant ownership on future views in schema FINANCE
to role ENV_FINANCE_FULL;
grant ownership on future stages in schema FINANCE
to role ENV_FINANCE_FULL;
grant ownership on future file formats in schema FINANCE
to role ENV_FINANCE_FULL;
grant ownership on future streams in schema FINANCE
to role ENV_FINANCE_FULL;
grant ownership on future procedures in schema FINANCE
to role ENV_FINANCE_FULL;
grant ownership on future functions in schema FINANCE
to role ENV_FINANCE_FULL;
grant ownership on future sequences in schema FINANCE
to role ENV_FINANCE_FULL;
```

Similarly as we have done for the FINANCE schema, the section above should be repeated for each schema in the database. The commands can be parametrized with the schema name and can be converted into a stored procedure for reusability.

Another recommendation is to review the object types for which we have

granted access. We have considered tables, external tables, views, stages, file formats, streams, procedures, functions, and sequences, but this list can be customized depending on the object types that are expected to be created by the users.

We will create functional roles using the environment role administrator ENV_ROLE_ADMIN:

```
use role ENV_ROLE_ADMIN;

-- create functional roles
create role ENV_DATA_SCIENTIST;
create role ENV_DATA_ANALYST;
create role ENV_DEVOPS;

grant role ENV_DATA_SCIENTIST to role ENV_SYS_ADMIN;
grant role ENV_DATA_ANALYST to role ENV_SYS_ADMIN;
grant role ENV_DEVOPS to role ENV_SYS_ADMIN;
```

As usual, we have granted all newly created roles to the environment system administrator ENV_SYS_ADMIN.

Finally, we will grant access roles to functional roles. We will allow the ENV_DEVOPS role to have full access on all objects. The ENV_DATA_SCIENTIST role will have read and write access to all objects, while the ENV_DATA_ANALYST role will have only read access:

```
use role ENV_ROLE_ADMIN;
```

```
-- grant access roles to functional roles
grant role ENV_FINANCE_READ to role ENV_DATA_ANALYST;
grant role ENV_FINANCE_READWRT to role ENV_DATA_SCIENTIST;
grant role ENV_FINANCE_FULL to role ENV_DEVOPS;
```

Once all of the above is done, we can grant functional roles to users. To test if the roles are working as expected, we can just grant them to ourselves for now (to the current user stored in the variable `my_current_user` that we have defined earlier):

```
grant role ENV_DATA_ANALYST to user identifier($my_current_user);
grant role ENV_DATA_SCIENTIST to user identifier($my_current_user);
grant role ENV_DEVOPS to user identifier($my_current_user);
```

We will also need a warehouse to execute queries, so let's quickly create a default warehouse using the SYSADMIN role and grant usage on the warehouse to the functional roles:

```
use role SYSADMIN;
create warehouse ENV_WH;
grant usage on warehouse ENV_WH to role ENV_DATA_SCIENTIST;
grant usage on warehouse ENV_WH to role ENV_DATA_ANALYST;
grant usage on warehouse ENV_WH to role ENV_DEVOPS;
```

We can try using one of the roles, for example ENV_DEVOPS. Using this role we will create a table named CUSTOMER:

```
use role ENV_DEVOPS;
create table CUSTOMER (cust_id integer, cust_name varchar);
```

We can check that the table was created in the Snowflake user interface or by executing the SHOW TABLES command. We will notice that the owner of the table is the ENV_FINANCE_FULL role, even though the role that created the table was ENV_DEVOPS. This is the expected behavior since we are using a managed access schema, where the role that has ownership privilege on the schema becomes the owner of the objects.

Still using the ENV_DEVOPS role, let's insert some data into the CUSTOMER table:

```
insert into CUSTOMER values (1, 'First Name');
```

The above statement should execute successfully because the ENV_DEVOPS role has the ENV_FINANCE_FULL role granted which owns the objects. This means that it can write data.

We can now switch to the DEV_DATA_ANALYST role and select from the CUSTOMER table:

```
use role DEV_DATA_ANALYST;
select * from CUSTOMER;
```

The select statement should succeed, because the DEV_DATA_ANALYST role has read access. However, if we try to insert data into the CUSTOMER table using the DEV_DATA_ANALYST role, the INSERT statement will fail due to insufficient privileges.

Using Secondary Roles

Before Snowflake introduced secondary roles, it was cumbersome to define cross-environment roles. For example, if the DevOps engineer needed access to both the development and the test environments, we could grant them role DEV_DEVOPS to access the development environment and role TEST_DEVOPS to access the test environment. The DevOps engineer could then switch between the two roles depending on which environment they were working on.

In cases when the DevOps engineer needed access to both environments at the same time, for example to copy data from the development environment to the test environment or vice versa, an additional cross-environment role would have to be created. For example, we would create a new functional role named DEVOPS and grant both DEV_DEVOPS and TEST_DEVOPS to this new role. The DevOps user would then use the DEVOPS role for cross-environment development.

With Snowflake secondary roles this is no longer required. As long as user DevOps has both DEV_DEVOPS and TEST_DEVOPS roles granted, they can

set one of these roles as their primary role and the other as the secondary role. Keeping in mind that authorization to create objects comes from the primary role only, the primary role would be assigned according to the environment where objects are being created or data is written.

For example, to create a table named CUSTOMER in the test environment by cloning the table from the development environment, the DevOps engineer would set the roles as follows:

```
use role TEST_DEVOPS;
use secondary roles DEV_DEVOPS;

create table TEST_DWH.FINANCE.CUSTOMER
clone DEV_DWH.FINANCE.CUSTOMER;
```

Other use cases for using secondary roles in cross-environment development would be to access source data from a source database and write the data to one of the development, test, UAT, or production target databases. Again, the user would use the primary role according to the database where they want to create objects or write data, and one or more secondary roles for each of the source databases.

Rbac | Role Based Access Control | Data Superhero

Responses (1)

PAVAN TEJA

What are your thoughts?

H

Harishrajamohan

Dec 27, 2022

Nice blog explaining in detail about the RBAC...

I have couple of questions, hope you could clarify.

1) grant all privileges on schema FINANCE to role ENV_FINANCE_FULL; in this statement, when we say all privileges what are the list of privileges does... more

1 reply Reply