Column Level Security and Dynamic Data Masking

Summary	In this codelab, we will learn how to create Synthetic Data using Snowflake
URL	https://docs.snowflake.com/en/user-guide/security-column.html
Category	Web
Status	Published
Author	Nikhil Kohli

Data Governance using Dynamic Data Masking

Data Governance using Dynamic Data Masking

Managing access to sensitive data is very important when it comes to security and data governance

Traditional role-based access control (RBAC) can be used to enforce and control least-privilege access. But what happens when traditional RBAC controls don't allow for the right flexibility? But Snowflake uses the new Dynamic Data Masking feature to design an RBAC model which gives us granular control over our employees' viewing permissions to support our data governance model.

ROLE-BASED ACCESS CONTROL

Snowflake offers a robust RBAC system where administrators can create custom roles to meet their organization's requirements. Snowflake's internal RBAC design is based on the principle of least privilege, where users are granted access to only the privileges required for their job duties. Roles are designed to create a clear separation of duties between read-only access and write access. Snowflake distinguishes this separation between enterprise user roles (read-only access) and administrative roles (write access).

However, even though Snowflake's RBAC model provided a solid framework to solve most access control requirements, it still presented limitations:

RBAC worked well to control write access to data, but read-only access offered a different set of challenges.
If roles and permissions are too granular, role management becomes very manual and unruly.
If roles and permissions are too broad, they do not adhere to the least-privilege model.
Managing access to sensitive data using the RBAC model meant that we had to require and track special approvals for each access request.

DATA MASKING TO THE RESCUE

Snowflake recently released the Dynamic Data Masking feature, which allows a designated administrator to create and apply column-level masking policies. These policies can be defined

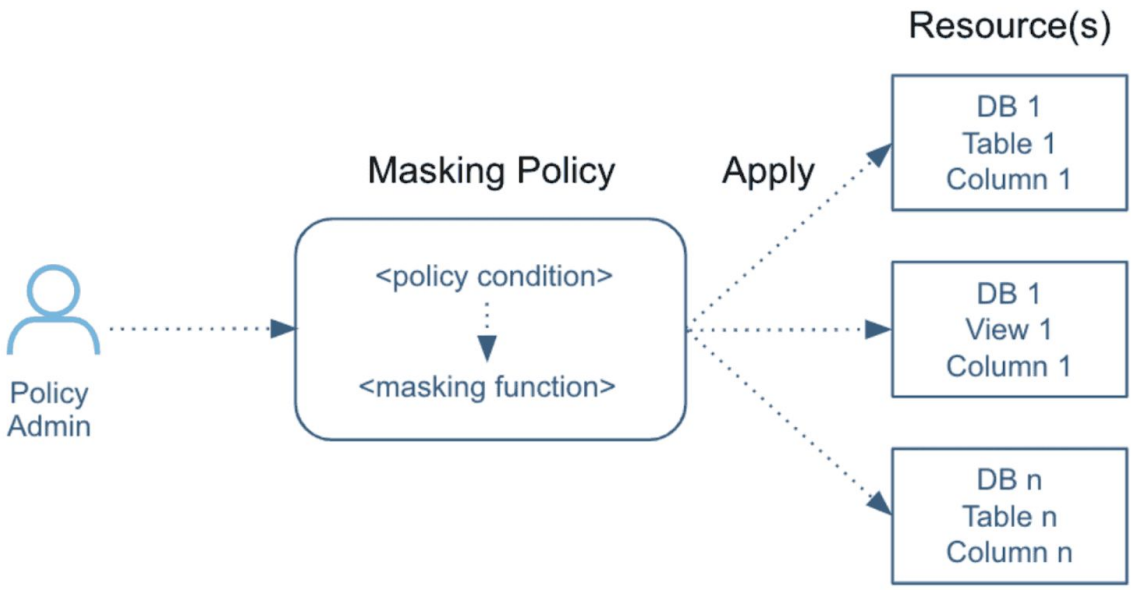


Figure 1: Policies can be defined to restrict access to data in the columns of tables or views.

specifically to restrict access to data in the columns (of tables or views) on which the policy is applied. Based on the policy, certain authorized roles (green-lit roles) will see the column values "as is," while the other roles (red-lit roles) will see obfuscated values. The definition of the policy also allows for much flexibility. For example, if you don't want to elaborate the names of every role that shouldn't see the exact value of the column cell, you can name only green-lit or red-lit roles. Users with the red-lit roles would still be able to query the table or view but would not see the actual values of the masked column.

We could create a set of policies to mask every column with sensitive data across an entire account. Only authorized roles assigned to the individuals who required access would be able to see those columns, and others would just see the value defined in the policy (for example, "000000"). The best part was that applying the new data masking policy did not negatively affect or break any queries or reports. Nothing changed for the majority of users.

Column level Security & Masking Policies

Column level Security & Masking Policies

Column-level security in Snowflake allows the application of a masking policy to a column within a table or view. Currently, column-level security includes two features:

1. Dynamic Data Masking

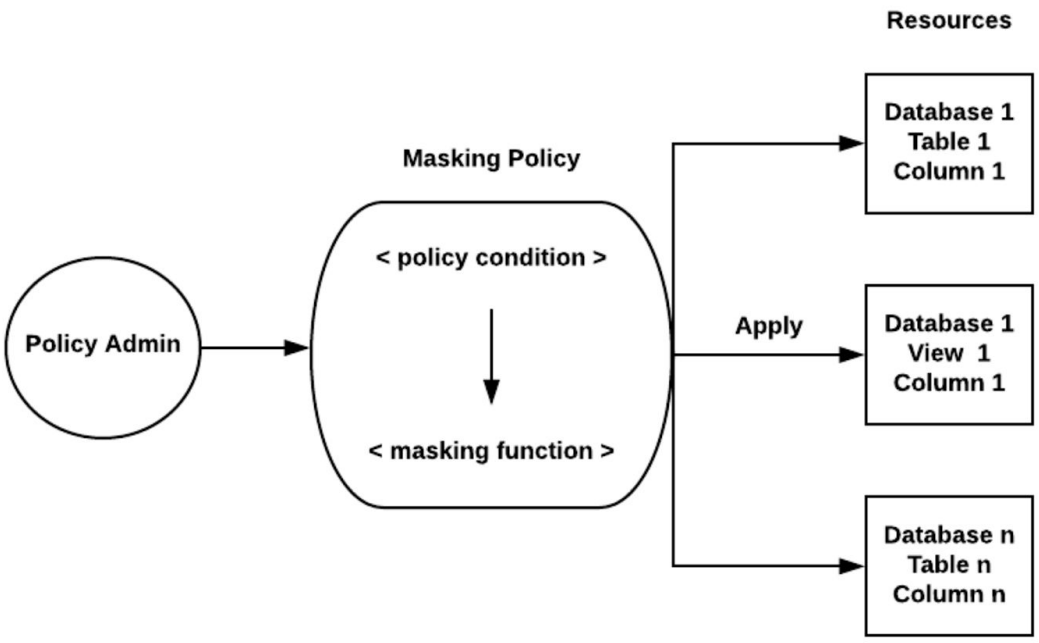
2. External Tokenization

Dynamic Data Masking is a column-level security feature that uses masking policies to selectively mask plain-text data in table and view columns at query time.

External Tokenization enables accounts to tokenize data before loading it into Snowflake and detokenize the data at query runtime. Tokenization is the process of removing sensitive data by replacing it with an undecipherable token. External Tokenization makes use of masking policies with external functions.

What are Masking Policies?

Snowflake supports masking policies as a schema-level object to protect sensitive data from unauthorized access while allowing authorized users to access sensitive data at query runtime. This means that sensitive data in Snowflake is not modified in an existing table (i.e. no static masking). Rather, when users execute a query in which a masking policy applies, the masking policy conditions determine whether unauthorized users see masked, partially masked, obfuscated, or tokenized data.



For example, masking policy administrators can implement a masking policy such that analysts (i.e. users with the custom ANALYST role) can only view the last four digits of a phone number and none of the social security number, while customer support representatives (i.e. users with the custom SUPPORT role) can view the entire phone number and social security number for customer verification use cases.

Authorized role (i.e. SUPPORT)

ID	Phone	SSN
101	408-123-5534	387-78-3456
102	510-334-3564	226-44-8908
103	214-553-9787	359-9987-0098

Unauthorized role (i.e. ANALYST)

ID	Phone	SSN
101	***-**-5534	********
102	***-**-3564	********
103	***-**-9787	********

Create Masking Policy

Create Masking Policy

```
-- Dynamic Data Masking

create masking policy employee_ssn_mask as (val string) returns string ->
  case
    when current_role() in ('PAYROLL') then val
    else '******'
  end;

-- External Tokenization

  create masking policy employee_ssn_detokenize as (val string) returns
string ->
  case
    when current_role() in ('PAYROLL') then ssn_unprotect(val)
    else val -- sees tokenized data
  end;
```

Where:

employee_ssn_mask
The name of the Dynamic Data Masking policy.

employee_ssn_detokenize
The name of the External Tokenization policy.

as (val string) returns string
Specifies the input and output data types. The data types must match.

->
Do not replace the arrow in the masking policy statement with any other character(s).

case ... end;
Specifies the masking policy body (i.e. SQL expression) conditions.

In these two examples, if the query operator is using the **PAYROLL** custom role in the current session, the operator sees the unmasked/de-tokenized value. Otherwise, a fixed masked/tokenized value is seen.

Using Dynamic Data Masking

Using Dynamic Data Masking

The following lists the high-level steps to configure and use Dynamic Data Masking in Snowflake:

1. Grant masking policy management privileges to a custom role for a security or privacy officer.

2. The security or privacy officer creates and defines masking policies and applies them to columns with sensitive data.

3. Execute queries in Snowflake. Note the following:

Snowflake dynamically rewrites the query applying the masking policy SQL expression to the column.
The column rewrite occurs at every place where the column specified in the masking policy appears in the query

Step 1: Grant Masking Policy Privileges to Custom Role

A security or privacy officer should serve as the masking policy administrator (i.e. custom role: MASKING_ADMIN) and have the privileges to define, manage, and apply masking policies to columns.

Privilege	Description
CREATE MASKING POLICY	This schema-level privilege controls who can create masking policies.
APPLY MASKING POLICY	This account-level privilege controls who can [un]set masking policies on columns and is granted to the ACCOUNTADMIN role by default. This privilege only allows applying a masking policy to a column and does ***not*** provide any additional table privileges described in Access Control Privileges.
APPLY ON MASKING POLICY	Optional. This policy-level privilege can be used by a policy owner to decentralize the [un]set operations of a given masking policy on columns to the object owners (i.e. the role that has the OWNERSHIP privilege on the object). Snowflake supports discretionary access control where object owners are also considered data stewards. If the policy administrator trusts the object owners to be data stewards for protected columns, then the policy administrator can use this privilege to decentralize applying the policy [un]set operations.

The following example creates the MASKING_ADMIN role and grants masking policy privileges to that role

```
-- create a masking policy administrator custom role

create role masking_admin;

-- grant privileges to masking_admin role.

grant create masking policy on schema <schema_name> to role masking_admin;

grant apply masking policy on account to role masking_admin;

-- allow table_owner role to set or unset the ssn_mask masking policy
(optional)

grant apply on masking policy ssn_mask to role table_owner;
```

Step 2: Create a Masking Policy and Apply to a Column

Using the MASKING_ADMIN role, create a masking policy and apply it to a column.

In this representative example, users with the ANALYST role see the unmasked value. Users without the ANALYST role see a full mask.

```
-- create masking policy

create or replace masking policy email_mask as (val string) returns string ->
  case
    when current_role() in ('ANALYST') then val
    else '*********'
  end;
```

Step 3: Apply the Masking Policy to a Table or View Column

Execute the following statements to apply the policy to a table column or a view column.

```
-- apply masking policy to a table column

alter table if exists user_info modify column email set masking policy
email_mask;

-- apply the masking policy to a view column

alter view user_info_v modify column email set masking policy email_mask;
```

Step 4: Query Data in Snowflake

Execute two different queries in Snowflake, one query with the ANALYST role and another query with a different role, to verify that users without the ANALYST role see a full mask.

```
-- using the ANALYST role

use role analyst;
select email from user_info; -- should see plain text value

-- using the PUBLIC role
```

```
use role public;
select email from user_info; -- should see full data mask
```

Additional Masking Policy Examples

Return NULL for unauthorized users:

```
case
  when current_role() IN ('ANALYST') then val
  else NULL
end;
```

Return a static masked value for unauthorized users:

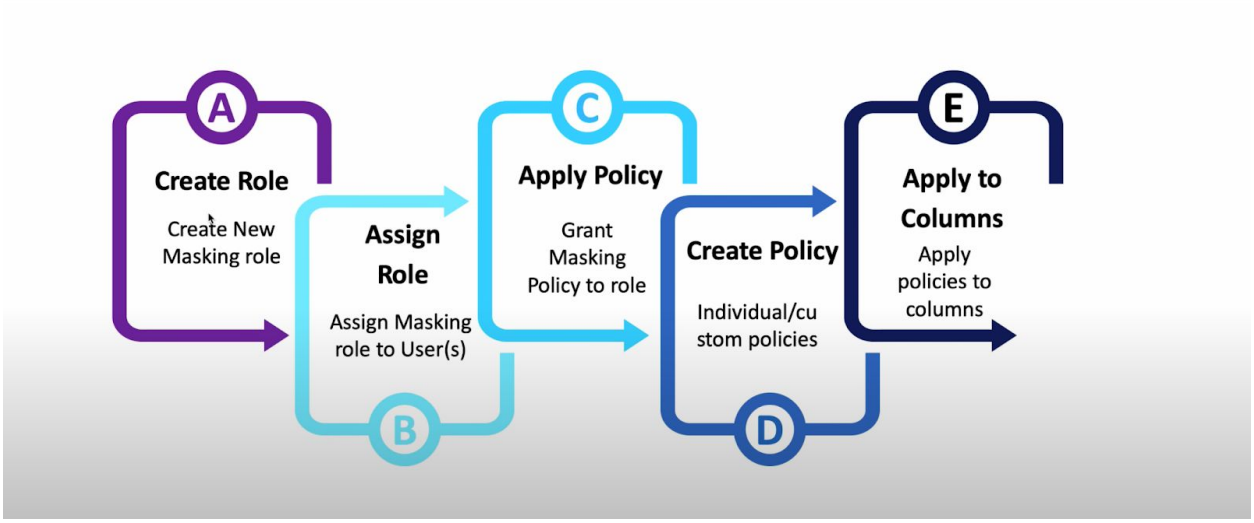
```
case
  when current_role() in ('ANALYST') then val
  else '********'
end;
```

Return a hash value using SHA2 , SHA2_HEX for unauthorized users:

```
case
  when current_role() in ('ANALYST') then val
  else sha2(val) -- return hash of the column value
```

```
end;
```

Hands-on Example on Dynamic Data Masking

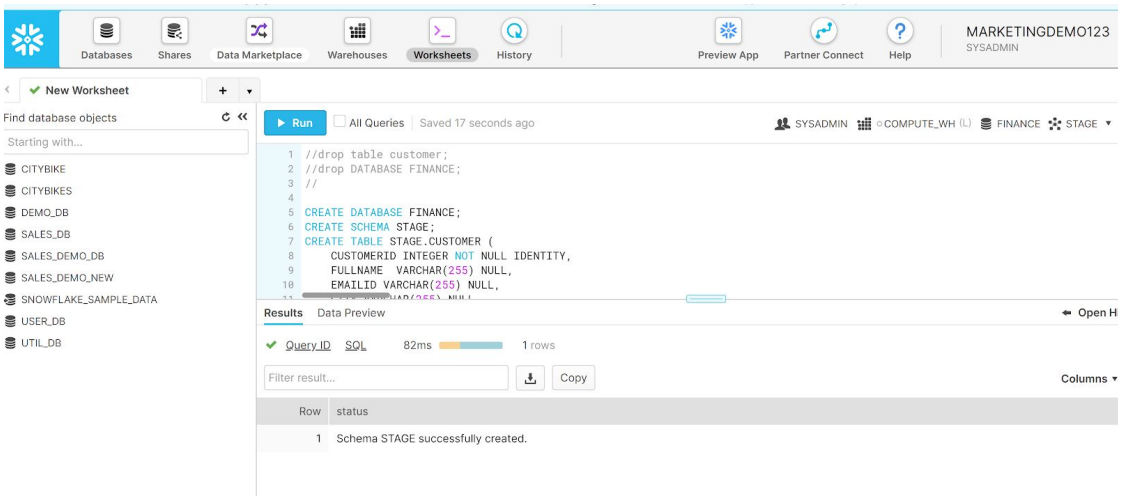


Run the file - Customer_TableData.sql or below code in snowflake

```
//drop table customer;
//drop DATABASE FINANCE;
//

CREATE DATABASE FINANCE;
CREATE SCHEMA STAGE;
CREATE TABLE STAGE.CUSTOMER (
    CUSTOMERID INTEGER NOT NULL IDENTITY,
    FULLNAME  VARCHAR(255) NULL,
    EMAILID VARCHAR(255) NULL,
    CITY VARCHAR(255) NULL,
    SSN VARCHAR(13) NULL,
```

```
    CREDITCARD VARCHAR(255) NULL,
    DATEOFBIRTH VARCHAR(255),
    PRIMARY KEY (CUSTOMERID)
);


INSERT INTO CUSTOMER( FULLNAME , EMAILID , CITY , SSN , CREDITCARD ,
DATEOFBIRTH ) VALUES('Chadwick Long','libero@vel.org','Bontang','16700219
2099','372619208674850','10/03/1959'),('Barry
White','ultrices.iaculis.odio@atvelitPellentesque.org','Gravelbourg','16631019
8236','341452321269970','11/28/1982'),('Nero
Johnston','molestie@dictumeleifend.net','Villa Verde','16930502
2155','372770250004239','04/22/1933'),('Thaddeus
Hood','diam.dictum@Etiam.org','Habra','16220325 9524','3481 458458
91872','12/25/1994'),('Keegan Page','per.conubia.nostra@mi.org','Puerto
Guzmán','16651018 0828','377643684865746','02/15/1997'),('Jordan
Gibson','inceptos@tellusSuspendissesed.org','Gudivada','16330423 6320','3750
277008 54524','11/01/1992'),('Gage
Mcmillan','in.consequat.enim@congueelitsed.com','Edmonton','16020722
2787','3403 873493 77002','06/03/1957'),('Andrew
Leblanc','semper@placerat.co.uk','Herk-de-Stad','16551204 9718','3444 276445
62852','11/30/1989'),('Brent
Clements','tempor@ipsumportaelit.co.uk','Lochristi','16430309
3423','377575623111156','10/11/2001'),('Steven
Hooper','imperdiet.dictum.magna@cursus.org','South Burlington','16231108
5886','3429 665990 48039','06/22/1992');



SELECT COUNT (*) FROM CUSTOMER;

SELECT * FROM CUSTOMER;
```

It will create the Database, Schema and Table and insert 100 records into the table

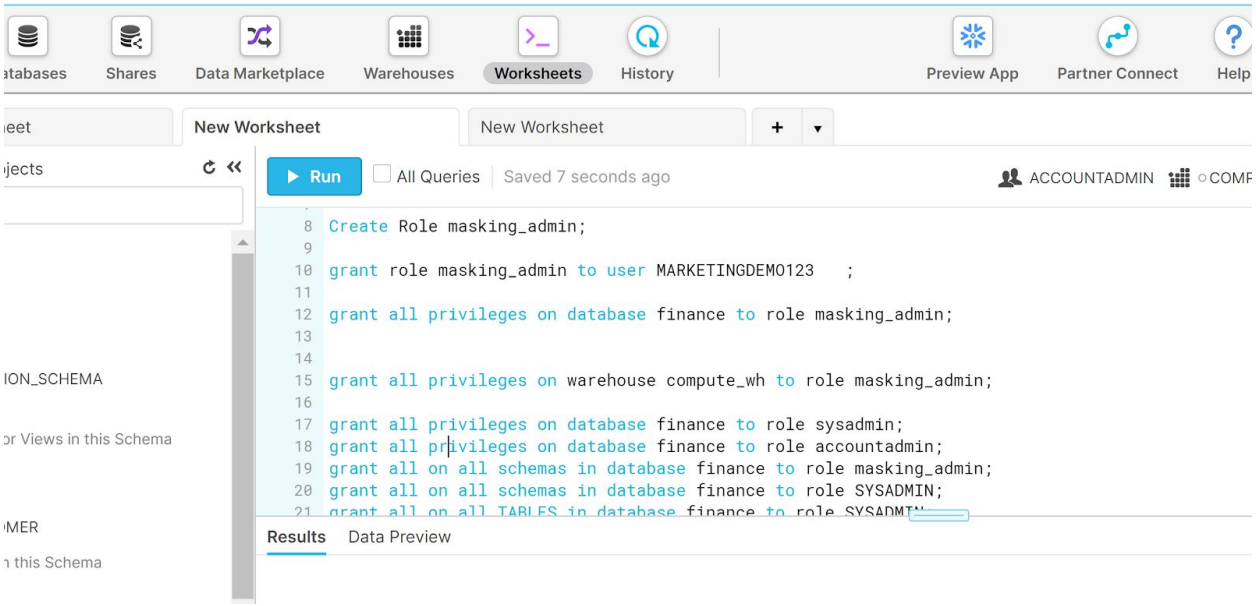


Create a new role - **Masked Admin** and grant access to Db and Schema

Assign this role to the user you want.

Grant Access to **Create & Apply masking policy** to the role masking admin.

```
USE FINANCE;

SELECT * FROM CUSTOMER;
Create user PII_USER password=piiuser default_role = masking_admin must_change_password = FALSE;


Create Role masking_admin;

grant role masking_admin to user PII_USER;

grant all privileges on database finance to role masking_admin;


grant all privileges on warehouse compute_wh to role masking_admin;

grant all privileges on database finance to role sysadmin;
grant all privileges on database finance to role accountadmin;
grant all on all schemas in database finance to role masking_admin;
grant all on all schemas in database finance to role SYSADMIN;
grant all on all TABLES in database finance to role SYSADMIN;
```



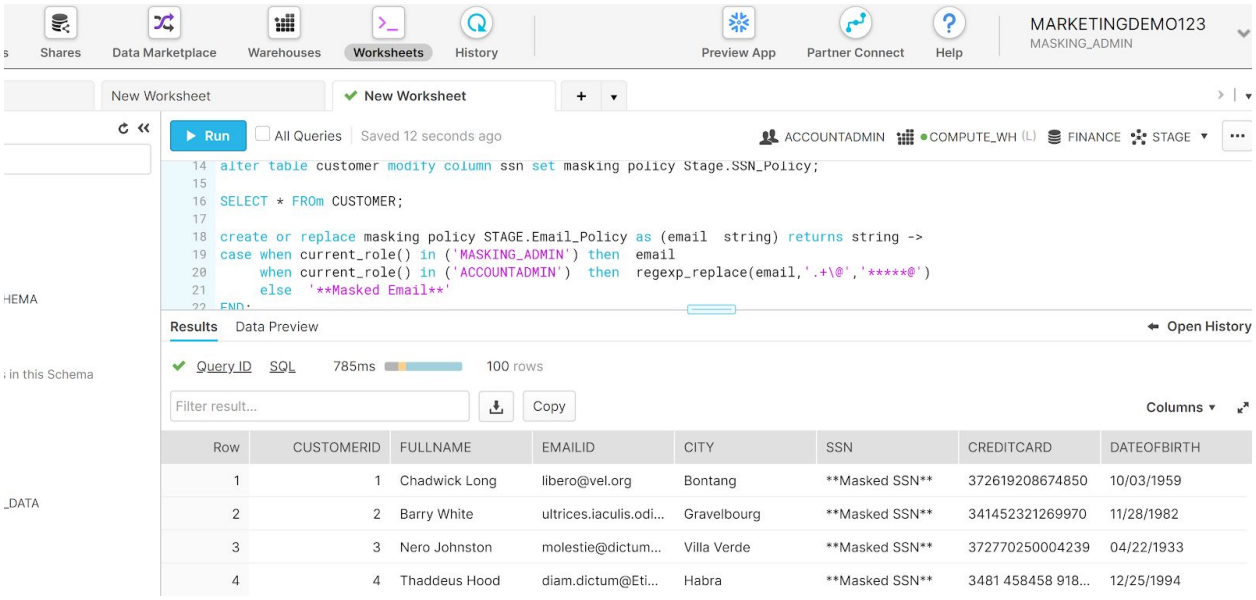
Create a new Masking policy for SNN/Email and associate it with a role(green lit roles or red lit)

Apply this masking policy to any Table/views column

```
--Create a new policy

create or replace masking policy STAGE.SSN_Policy as (SSN  string) returns string ->
case when current_role() in ('MASKING_ADMIN') then
        SSN
    ELSE
        '**Masked SSN**'
END;


---Associate it
alter table customer modify column ssn set masking policy Stage.SSN_Policy;

SELECT * FROm stage.CUSTOMER;
```

Query the data from different users and roles to see the output

SNN for AccountAdmin



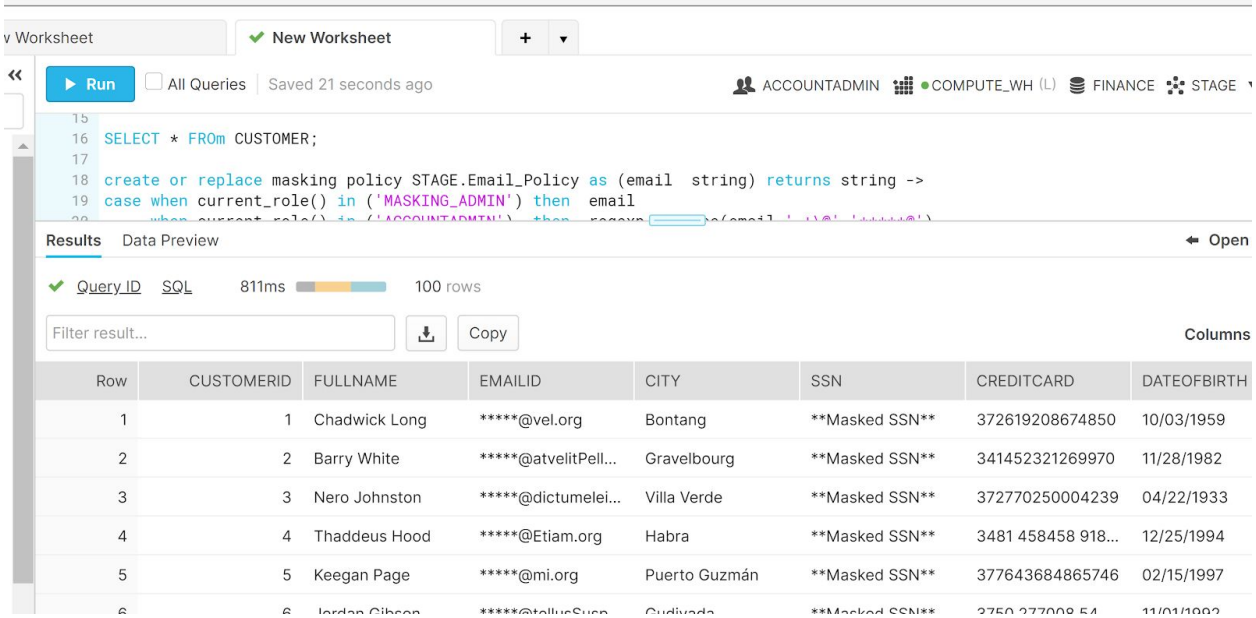
You can see the Masked SSN value for the Account Admin for SSN column.
Once the policy is created it can be easily extended to 100s of similar columns in different databases.

Email Masking

Same way you can create the policy for the Email where to one role we will do full masking and for the other partial masking

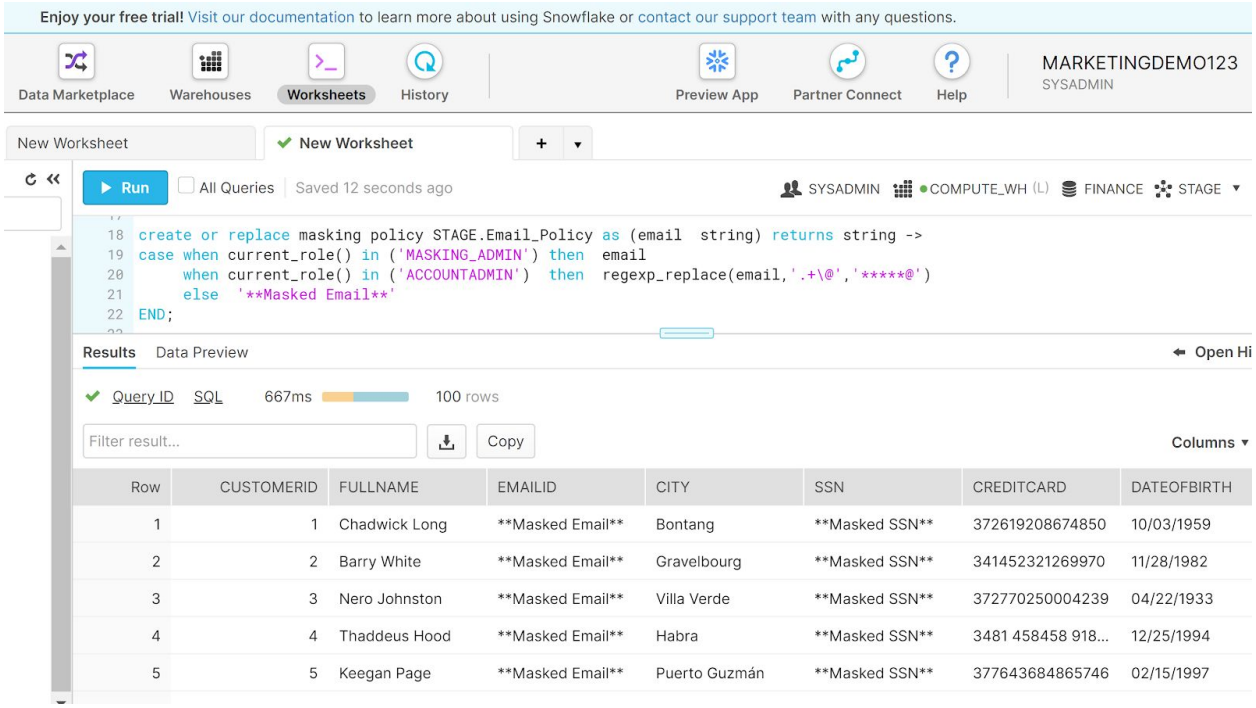
```
create or replace masking policy STAGE.Email_Policy as (email  string) returns string ->
case when current_role() in ('MASKING_ADMIN') then  email
     when current_role() in ('ACCOUNTADMIN')  then  regexp_replace(email,'.+\@','*****@')
     else  '**Masked Email**'
END;


alter table customer modify column emailid set masking policy Stage.Email_Policy;

SELECT * FROm stage.CUSTOMER;
```

AccountAdmin Role



Row	CUSTOMERID	FULLNAME	EMAILID	CITY	SSN	CREDITCARD	DATEOFBIRTH
1	1	Chadwick Long	*****@vel.org	Bontang	**Masked SSN**	372619208674850	10/03/1959
2	2	Barry White	*****@atvelitPell...	Gravelbourg	**Masked SSN**	341452321269970	11/28/1982
3	3	Nero Johnston	*****@dictumelei...	Villa Verde	**Masked SSN**	372770250004239	04/22/1933
4	4	Thaddeus Hood	*****@Etiam.org	Habra	**Masked SSN**	3481 458458 918...	12/25/1994
5	5	Keegan Page	*****@mi.org	Puerto Guzmán	**Masked SSN**	377643684865746	02/15/1997

Sysadmin Role



Row	CUSTOMERID	FULLNAME	EMAILID	CITY	SSN	CREDITCARD	DATEOFBIRTH
1	1	Chadwick Long	**Masked Email**	Bontang	**Masked SSN**	372619208674850	10/03/1959
2	2	Barry White	**Masked Email**	Gravelbourg	**Masked SSN**	341452321269970	11/28/1982
3	3	Nero Johnston	**Masked Email**	Villa Verde	**Masked SSN**	372770250004239	04/22/1933
4	4	Thaddeus Hood	**Masked Email**	Habra	**Masked SSN**	3481 458458 918...	12/25/1994
5	5	Keegan Page	**Masked Email**	Puerto Guzmán	**Masked SSN**	377643684865746	02/15/1997