

Object Tagging

This topic provides concepts and instructions on how to use tags in Snowflake.

To learn more about using a masking policy with a tag, see [Tag-based masking policies](#).

ENTERPRISE EDITION FEATURE

This feature requires Enterprise Edition or higher. To inquire about upgrading, please contact [Snowflake Support](#).

What is a tag?

Tags enable data stewards to monitor sensitive data for compliance, discovery, protection, and resource usage use cases through either a centralized or decentralized data governance management approach.

A tag is a schema-level object that can be assigned to another Snowflake object. A tag can be assigned an arbitrary string value upon assigning the tag to a Snowflake object. Snowflake stores the tag and its string value as a key-value pair. The tag must be unique for your schema, and the tag value is always a string.

You create a tag using a [CREATE TAG](#) statement, and you specify the tag string value when assigning the tag to an object. The tag can be assigned to an object while creating the object, using a [CREATE <object>](#) statement, assuming that the tag already exists. Alternatively, you can assign the tag to an existing object using an [ALTER <object>](#) statement.

A single tag can be assigned to different object types at the same time (e.g. warehouse and table simultaneously). At the time of assignment, the tag string value can be duplicated or remain unique. For example, multiple tables can be assigned the `cost_center` tag and the tag can always have the string value be `sales`. Alternatively, the string value could be different (e.g. `engineering`, `marketing`, `finance`). After defining the tags and assigning the tags to Snowflake objects, tags can be queried to monitor usage on the objects to facilitate data governance operations, such as monitoring, auditing, and reporting.

Because tags can be assigned to tables, views, and columns, setting a tag and then querying the tag enables the discovery of a multitude of database objects and columns that contain sensitive information. Upon discovery, data stewards can determine how best to make that data available, such as selective filtering using [row access policies](#), or using [masking policies](#) to determine whether the data is tokenized, fully masked, partially masked, or unmasked.

Assigning tags to warehouses enables accurate resource usage monitoring. Querying tags on resources allows for easy resource grouping by cost center or other organization units. Additionally, the tag can facilitate analyzing relatively short-term business activities, such as projects, to provide a more granular insight into what, when, and how resources were used.

To begin, let's create some sample resources. We will create three databases that we will leverage.

```
-- Create sample databases|

USE ROLE ACCOUNTADMIN;

CREATE OR REPLACE DATABASE SNOWTEST; -- Database storing the tags
CREATE OR REPLACE DATABASE SNOWTEST2;
CREATE OR REPLACE DATABASE SNOWTEST3;
```

Next, we will create tags:

```
-- We need to create a schema for the object tags

CREATE OR REPLACE SCHEMA SNOWTEST.TAGS;

-- Let's create a tag for the different database objects

CREATE OR REPLACE TAG SNOWTEST.TAGS.tag_options allowed_values 'sales','finance';

-- We can look at the tags
SELECT GET_DDL('tag', 'tag_options');
```

Results		Chart
	GET_DDL('TAG', 'TAG_OPTIONS')	
1	create or replace tag TAG_OPTIONS allowed_values 'finance', 'sales';	

We can add the tags using the ALTER command

```
-- We can Tag objects by using the ALTER command and SET TAG


ALTER DATABASE SNOWTEST2 SET TAG tag_options = 'sales';

-- We can ONLY add tags where there is an available option

ALTER DATABASE SNOWTEST3 SET TAG tag_options = 'IT'; -- Should display an error
```

Results

Chart



Value 'IT' is not allowed by the specified allowed_values for tag 'TAG_OPTIONS'.

We can alter the tag to add the IT option; thus enable us to add the tag:

```
!
-- ALTER the Tag to add the IT Option

ALTER TAG SNOWTEST.TAGS.tag_options add allowed_values 'IT';
ALTER DATABASE SNOWTEST3 SET TAG SNOWTEST.TAGS.tag_options = 'IT';

-- We can always see all the available options for a tag in Snowflake

| SELECT SYSTEM$GET_TAG_ALLOWED_VALUES( 'SNOWTEST.TAGS.TAG_OPTIONS' );

-- Furthermore, we can see what objects have various tags
```

Results		Chart
	SYSTEM\$GET_TAG_ALLOWED_VALUES('SNOWTEST.TAGS.TAG_OPTIONS')	
1	["IT","finance","sales"]	

Introduction to Classification

ENTERPRISE EDITION FEATURE

This feature requires Enterprise Edition or higher. To inquire about upgrading, please contact [Snowflake Support](#).

This topic provides information on how classification works.

For information on how to use custom classifiers, see [Custom Data Classification](#).

Overview

Classification is a multi-step process that associates Snowflake-defined system tags to columns by analyzing the fields and metadata for personal data; this data can be tracked by a data engineer using SQL and Snowsight. A data engineer can classify columns in a table to determine whether the column contains certain kinds of data that need to be tracked or protected, such a unique identifier (passport or bank account data), a quasi-identifier (the city in which the individual lives), or a sensitive value (the salary of an individual).

By tracking the data with a system tag and protecting the data by using a masking or row access policy, the data engineer can improve the governance posture associated with the data. The overall result of the classification and data protection steps is to facilitate compliance with data privacy regulations.

You can classify a single table or tables in a schema. Snowflake provides predefined [system tags](#) to enable you to classify and tag columns, or you can use custom classifiers to define your own semantic category based on your knowledge of your data. You can also choose an approach the uses Snowflake system tags and custom classifiers depending on the governance posture that you wish to adopt.

Classification provides the following benefits to data privacy and data governance administrators:

- | | |
|---------------------------|--|
| Data access | The results of classifying column data can inform identity and access management administrators to evaluate and maintain their Snowflake role hierarchies to ensure the Snowflake roles have the appropriate access to sensitive or PII data. |
| Data sharing | The classification process can help to identify and confirm the storage location of PII data. Subsequently, a data sharing provider can use the classification results to determine whether to share data and how to make the PII data available to a data sharing consumer. |
| Policy application | The usage of columns containing PII data, such as referencing columns in base tables to create a view or materialized view, can help to determine the best approach to protect the data with either a masking policy or a row access policy. |

System tags and categories

System tags are tags that Snowflake creates, maintains, and makes available in the shared [SNOWFLAKE database](#). There are two Classification system tags, both of which exist in the `SNOWFLAKE.CORE` schema:

- `SNOWFLAKE.CORE.SEMANTIC_CATEGORY`
- `SNOWFLAKE.CORE.PRIVACY_CATEGORY`

The data engineer assigns these tags to a column containing personal or sensitive data.

String values Snowflake stores the assignment of a system tag on a column as a key-value pair, where the value is a string. Snowflake defines the allowed string values for each classification system tag because Snowflake maintains each of these system tags.

The tag names, `SEMANTIC_CATEGORY` and `PRIVACY_CATEGORY`, correspond to the Classification categories that Snowflake assigns to the column data during the column sampling process (i.e. the tag names and category names use the same words):

Semantic category The semantic category identifies personal attributes.

A non-exhaustive list of personal attributes Classification supports include name, age, and gender. These three attributes are possible string values when assigning the `SEMANTIC_CATEGORY` tag to a column.

Classification can detect information from different countries, such as Australia, Canada, and the United Kingdom. For example, if your table column contains phone number information, the analysis process can differentiate the different phone number values from each of these countries.

Privacy category If the analysis determines that the column data corresponds to a semantic category, Snowflake further classifies the column to a privacy category. The privacy category has three values: identifier, quasi-identifier, or sensitive. These three values are the string values that can be specified when assigning the `PRIVACY_CATEGORY` Classification system tag to a column.

- Identifier: These attributes uniquely identify an individual. Example attributes include name, social security number, and phone number.

Identifier attributes are synonymous with *direct identifiers*.

- Quasi-identifier: These attributes can uniquely identify an individual when two or more of these attributes are in combination. Example attributes include age and gender.

Quasi-identifiers are synonymous with *indirect identifiers*.

- Sensitive: These attributes are not considered enough to identify an individual but are information that the individual would rather not disclose for privacy reasons.

Currently, the only attribute that Snowflake evaluates as sensitive is salary.

- Insensitive: These attributes do not contain personal or sensitive information.

The following table summarizes the relationship between each classification category and system tag, and the string values for each classification system tag. Snowflake supports international SEMANTIC_CATEGORY tag values that pertain to certain countries. The country codes are based on the [ISO-3166-1 alpha-2](#) standard. Other semantic categories, such as `EMAIL` and `GENDER`, do not have a country code. To track international information, the data engineer uses the value in the SEMANTIC_CATEGORY tag values column when setting a system tag on a column.

PRIVACY_CATEGORY values	SEMANTIC_CATEGORY values	Supported countries
IDENTIFIER	<ul style="list-style-type: none"><code>BANK_ACCOUNT</code><code>DRIVERS_LICENSE</code><code>MEDICARE_NUMBER</code><code>NATIONAL_IDENTIFIER</code><code>ORGANIZATION_IDENTIFIER</code><code>PASSPORT</code><code>PHONE_NUMBER</code><code>STREET_ADDRESS</code><code>TAX_IDENTIFIER</code><code>EMAIL</code><code>IBAN</code><code>IMEI</code><code>IP_ADDRESS</code><code>NAME</code><code>PAYMENT_CARD</code><code>URL</code><code>VIN</code>	<ul style="list-style-type: none">CA, NZ, USAU, CA, NZ, USAU, NZCA, NZ, SG, UK, US (SSN)AU, NZ, SGAU, CA, NZ, SG, USAU, CA, JP, UK, USCA, USAU, NZ, US (EIN, ITIN)

QUASI_IDENTIFIER	• ADMINISTRATIVE_AREA_1	• CA, NZ, US
	• ADMINISTRATIVE_AREA_2	• US
	• CITY	• CA, NZ, US
	• POSTAL_CODE	• AU, CA, CH, JP, NZ, UK, US
	• AGE	
	• COUNTRY	
	• DATE_OF_BIRTH	
	• ETHNICITY	
	• GENDER	
	• LAT_LONG	
	• LATITUDE	
	• LONGITUDE	
	• MARITAL_STATUS	
	• OCCUPATION	
	• YEAR_OF_BIRTH	

SENSITIVE	• SALARY
-----------	----------

Let's create data classifications. First we will create a sample database.

```
-- Create a sample data and table with ID, Social Security number, Age and Credit Card

USE ROLE ACCOUNTADMIN;

CREATE OR REPLACE DATABASE SNOWTEST;
CREATE OR REPLACE SCHEMA SNOWTEST.DATA_CLASS;
CREATE OR REPLACE TABLE SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL (
    ID VARCHAR(10)
    , SSN VARCHAR(11)
    , AGE NUMERIC
    , CREDIT_CARD VARCHAR(19)
);
```



```
);

-- Let's enter some fake sensitive data
INSERT INTO SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL
VALUES ('A0000001', '234-45-6477', 24, '4053-0495-0394-0494'),
('A0000002', '234-85-3427', 28, '4653-0495-0394-0494'),
('A0000003', '235-49-6477', 43, '4053-0755-0394-0494'),
('A0000004', '254-85-6457', 57, '4653-4566-0394-0494'),
('A0000005', '235-45-6076', 34, '4053-0755-0394-0494'),
('A0000006', '454-85-6473', 21, '4653-0495-0394-0494'),
('A0000008', '285-49-6697', 17, '4053-0755-0394-0494'),
('A0000009', '234-85-7377', 12, '4653-0495-0394-0494'),
('A0000010', '253-45-6467', 87, '4053-0755-0394-0494'),
('A0000012', '434-85-6384', 58, '4653-0495-0394-0494'),
('A0000013', '893-45-4266', 34, '4053-0755-0394-0494'),
('A0000011', '684-85-9405', 74, '4653-0495-3454-0494'),
('A0000014', '345-45-6935', 28, '4053-0755-0394-0494'),
('A0000015', '034-85-6637', 53, '4653-0495-0394-0494'),
('A0000016', '475-45-5646', 34, '4053-0755-0394-0494'),
('A0000017', '694-85-6030', 38, '4653-0495-0394-0494'),
('A0000018', '745-45-5466', 36, '4053-0755-0394-0494'),
('A0000019', '224-85-3246', 74, '4653-0495-0394-0494'),
('A0000020', '945-45-9645', 45, '4053-0755-0394-0494');

-- Looking at the table values
SELECT *
FROM SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL;
```

```
364 -- Looking at the table values
365 SELECT *
366 FROM SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL;
367
368
```

Results				
	ID	SSN	AGE	CREDIT_CARD
1	A0000001	234-45-6477	24	4053-0495-0394-0494
2	A0000002	234-85-3427	28	4653-0495-0394-0494
3	A0000003	235-49-6477	43	4053-0755-0394-0494
4	A0000004	254-85-6457	57	4653-4566-0394-0494
5	A0000005	235-45-6076	34	4053-0755-0394-0494

Next we will use `EXTRACT_SEMANTIC_CATEGORIES` to extract the data categories. We will end up with a JSON object:

```
-- Run EXTRACT_SEMANTIC_CATEGORIES to analyze the columns in the table
SELECT EXTRACT_SEMANTIC_CATEGORIES('SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL');
```



```
[ ] EXTRACT_SEMANTIC_CATEGORIES('SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL')
{
  "AGE": {
    "alternates": [],
    "recommendation": {
      "confidence": "HIGH",
      "coverage": 1,
      "details": [],
      "privacy_category": "QUASI_IDENTIFIER",
      "semantic_category": "AGE"
    },
    "valid_value_ratio": 1
  },
  "CREDIT_CARD": {
    "alternates": []
  },
  "ID": {
    "alternates": []
  },
  "SSN": {
    "alternates": [],
    "recommendation": {
      "confidence": "HIGH",
      "coverage": 0.94736844,
      "details": [
        {
          "coverage": 0.94736844,
          "semantic_category": "US_SSN"
        }
      ],
      "privacy_category": "IDENTIFIER",
      "semantic_category": "NATIONAL_IDENTIFIER"
    }
  }
}
```

We can flatten the data and review the results in a table:

```

-- The results will be in JSON format. We can flatten to analysis the recommendations

CREATE OR REPLACE TABLE SNOWTEST.DATA_CLASS.CLASSIFICATIONS AS
select t.key::varchar as column_name,
       t.value:"recommendation".privacy_category::varchar as privacy_category,
       t.value:"recommendation".semantic_category::varchar as semantic_category,
       t.value:"recommendation".coverage::numeric as probability
from table(
    flatten(
        extract_semantic_categories(
            'SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL'
        )::variant
    )
) as t
;

-- We can then use this table to review the recommendation and create object tags accordingly
SELECT * FROM SNOWTEST.DATA_CLASS.CLASSIFICATIONS;

```

	COLUMN_NAME	PRIVACY_CATEGORY	SEMANTIC_CATEGORY	PROBABILITY
1	AGE	QUASI_IDENTIFIER	AGE	1
2	CREDIT_CARD	null	null	null
3	ID	null	null	null
4	SSN	IDENTIFIER	NATIONAL_IDENTIFIER	1

Understanding Column-level Security

ENTERPRISE EDITION FEATURE

This feature requires Enterprise Edition (or higher). To inquire about upgrading, please contact [Snowflake Support](#).

This topic provides a general overview of Column-level Security and describes the features and support that are common to both Dynamic Data Masking and External Tokenization.

What is Column-level Security?

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](#)

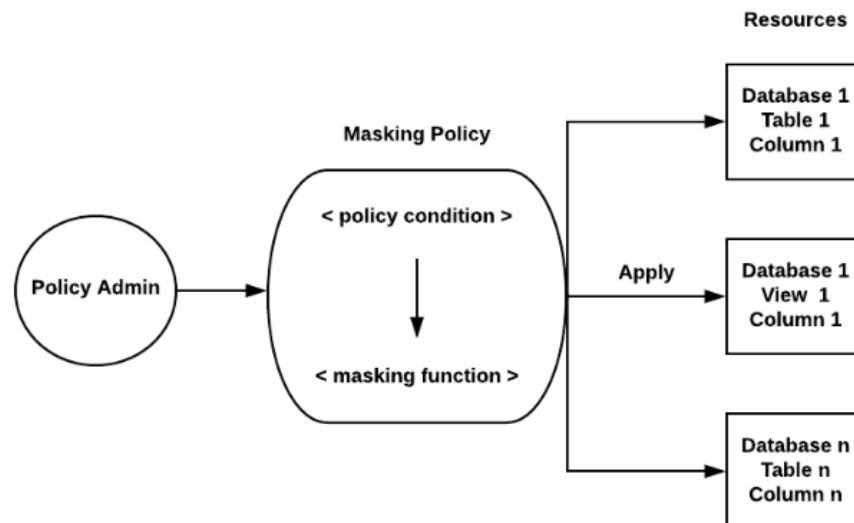
Dynamic Data Masking is a Column-level Security feature that uses masking policies to selectively mask plain-text data in table and view columns at query time.

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. Masking policies as a schema-level object also provide flexibility in choosing a centralized, decentralized, or hybrid management approach. For more information, see [Managing Column-level Security](#) (in this topic).

Masking policies can include conditions and functions to transform the data at query runtime when those conditions are met. The policy-driven approach supports segregation of duties to allow security teams to define policies that can limit sensitive data exposure, even to the owner of an object (i.e. the role with the OWNERSHIP privilege on the object, such as a table or view) who normally have full access to the underlying data.



Let's create a masking policy. To begin we will spin up a sample database.

```

-- Create a sample data and table with ID, Social Security number, Age and Credit Card

USE ROLE ACCOUNTADMIN;

CREATE OR REPLACE DATABASE SNOWTEST;
CREATE OR REPLACE SCHEMA SNOWTEST.DATA_CLASS;
CREATE OR REPLACE TABLE SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL(
    ID VARCHAR(10)
    , SSN VARCHAR(11)
    , AGE NUMERIC
    , CREDIT_CARD VARCHAR(19)
);

-- Let's enter some fake sensitive data

INSERT INTO SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL
VALUES ('A0000001', '234-45-6477', 24, '4053-0495-0394-0494'),
('A0000002', '234-85-3427', 28, '4653-0495-0394-0494'),
('A0000003', '235-49-6477', 43, '4053-0755-0394-0494'),
('A0000004', '254-85-6457', 57, '4653-4566-0394-0494'),
('A0000005', '235-45-6076', 34, '4053-0755-0394-0494'),
('A0000006', '454-85-6473', 21, '4653-0495-0394-0494'),
('A0000008', '285-49-6697', 17, '4053-0755-0394-0494'),
('A0000009', '234-85-7377', 12, '4653-0495-0394-0494'),
('A0000010', '253-45-6467', 87, '4053-0755-0394-0494'),
('A0000012', '434-85-6384', 58, '4653-0495-0394-0494'),
('A0000013', '893-45-4266', 34, '4053-0755-0394-0494'),
('A0000011', '684-85-9405', 74, '4653-0495-3454-0494'),
('A0000014', '345-45-6935', 28, '4053-0755-0394-0494'),
('A0000015', '034-85-6637', 53, '4653-0495-0394-0494'),
('A0000016', '475-45-5646', 34, '4053-0755-0394-0494'),
('A0000017', '694-85-6030', 38, '4653-0495-0394-0494'),
('A0000018', '745-45-5466', 36, '4053-0755-0394-0494'),
('A0000019', '224-85-3246', 74, '4653-0495-0394-0494'),
('A0000020', '945-45-9645', 45, '4053-0755-0394-0494');

-- Here is the table, but we have sensitive data like SSN and Credit Card information
SELECT * FROM SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL;

-- We want to mask the social security number so it only shows the last 4 digits
SELECT CONCAT('XXX-XX-', RIGHT(SSN, 4)) as SSN
FROM SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL;

```

↶ Results

↷ Chart

	SSN
1	XXX-XX-6477
2	XXX-XX-3427
3	XXX-XX-6477
4	XXX-XX-6457
5	XXX-XX-6076
6	XXX-XX-6473
7	XXX-XX-6697
8	XXX-XX-7377
9	XXX-XX-6467
10	XXX-XX-6384

Let's create a new role for analyst. We want the analyst to see the entire SSN number, where everyone else can only see the masked value:

```
-- Create a analyst role
```

```
CREATE OR REPLACE ROLE analyst;
```

```
GRANT ROLE analyst TO USER <user>;
```

```
GRANT ALL ON WAREHOUSE COMPUTE_WH TO ROLE analyst;
```

```
GRANT ALL ON DATABASE SNOWTEST TO ROLE analyst;
```

```
GRANT ALL ON SCHEMA SNOWTEST.DATA_CLASS TO ROLE analyst;
```

```
GRANT ALL ON TABLE SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL TO ROLE analyst;
```

Now we create the policy

```
-- Create masking policy for SSN number
```

```
CREATE OR REPLACE MASKING POLICY ssn_mask AS (val string) RETURNS string ->
```

```
  CASE
```

```
    WHEN CURRENT_ROLE() = 'ANALYST' THEN val
```

```
    ELSE CONCAT('XXX-XX-',RIGHT(val,4))
```

```
  END;
```

```
-- Now apply the masking policy on the sample table SSN column
```

```
ALTER TABLE IF EXISTS SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL MODIFY COLUMN SSN SET MASKING POLICY ssn_mask;
```

If we switch to analyst, we will see the entire SSN

```
-- using the ANALYST role

USE ROLE analyst;
SELECT CURRENT_ROLE();
SELECT * FROM SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL; -- should see plain text value
```

	ID	SSN	AGE	CREDIT_CARD
1	A0000001	234-45-6477	24	4053-0495-0394-0494
2	A0000002	234-85-3427	28	4653-0495-0394-0494
3	A0000003	235-49-6477	43	4053-0755-0394-0494
4	A0000004	254-85-6457	57	4653-4566-0394-0494
5	A0000005	235-45-6076	34	4053-0755-0394-0494
6	A0000006	454-85-6473	21	4653-0495-0394-0494
7	A0000008	285-49-6697	17	4053-0755-0394-0494
8	A0000009	234-85-7377	12	4653-0495-0394-0494
9	A0000010	253-45-6467	87	4053-0755-0394-0494
10	A0000010	434-85-6384	58	4653-0495-0394-0494

However, when we switch back to account admin, the masking policy will be applied

```
-- using the ACCOUNTADMIN role

USE ROLE ACCOUNTADMIN;
SELECT * FROM SNOWTEST.DATA_CLASS.SAMPLE_DATA_TBL; -- should see full data mask
```

ID	SSN	AGE	CREDIT_CARD
A0000001	XXX-XX-6477	24	4053-0495-0394-0494
A0000002	XXX-XX-3427	28	4653-0495-0394-0494
A0000003	XXX-XX-6477	43	4053-0755-0394-0494
A0000004	XXX-XX-6457	57	4653-4566-0394-0494
A0000005	XXX-XX-6076	34	4053-0755-0394-0494
A0000006	XXX-XX-6473	21	4653-0495-0394-0494
A0000008	XXX-XX-6697	17	4053-0755-0394-0494
A0000009	XXX-XX-7377	12	4653-0495-0394-0494
A0000010	XXX-XX-6467	87	4053-0755-0394-0494
A0000010	XXX-XX-6384	58	4653-0495-0394-0494

Test Your Skills

- Create three new roles, administration, enrollment and accounting
- Create a masking policy so that SSN and Credit Card is masked for administration, credit card is masked for enrollment and SSN is masked for finance

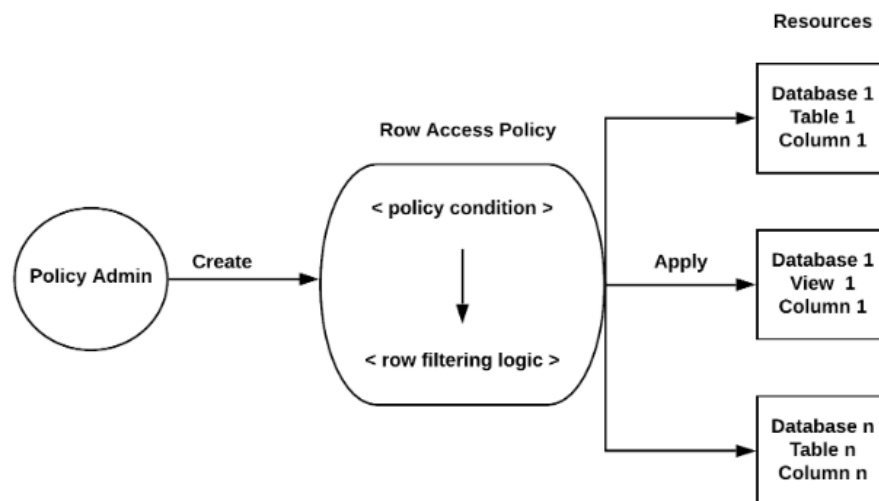
What is Row-level Security?

Snowflake supports row-level security through the use of row access policies to determine which rows to return in the query result. The row access policy can be relatively simple to allow one particular role to view rows, or be more complex to include a [mapping table](#) in the policy definition to determine access to rows in the query result. If the policy contains a mapping table lookup, create a centralized mapping table and store the mapping table in the same database as the protected table. This is particularly important if the policy calls the [IS_DATABASE_ROLE_IN_SESSION](#) function. For details, see the function usage notes.

A row access policy is a schema-level object that determines whether a given row in a table or view can be viewed from the following types of statements:

- [SELECT](#) statements
- Rows selected by [UPDATE](#), [DELETE](#), and [MERGE](#) statements.

Row access policies can include conditions and functions in the policy expression to transform the data at query runtime when those conditions are met. The policy-driven approach supports segregation of duties to allow governance teams to define policies that can limit sensitive data exposure. This approach also includes the object owner (i.e. the role with the OWNERSHIP privilege on the object, such as a table or view) who normally has full access to the underlying data. A single policy can be set on different tables and views at the same time.



Row access policies do not currently prevent rows from being inserted, or prevent visible rows from being updated or deleted.

To begin with Row level access, we will create a sample dataset

```

USE ROLE ACCOUNTADMIN;

CREATE OR REPLACE DATABASE SNOWTEST;
CREATE OR REPLACE SCHEMA SNOWTEST.PUBLIC;

-- Create roles and apply to the user

CREATE OR REPLACE ROLE ROLE1;
CREATE OR REPLACE ROLE ROLE2;
CREATE OR REPLACE ROLE ROLE3;
CREATE OR REPLACE ROLE SUPERADMIN;

GRANT ROLE SUPERADMIN TO USER <user>;
GRANT ROLE ROLE1 TO USER <user>;
GRANT ROLE ROLE2 TO USER <user>;
GRANT ROLE ROLE3 TO USER <user>;

CREATE OR REPLACE TABLE SNOWTEST.PUBLIC.SAMPLE_DATA_TBL (
    STATE VARCHAR(2)
    , SSN VARCHAR(11)
    , AGE NUMERIC
    , CC VARCHAR(19)
);

INSERT INTO SNOWTEST.PUBLIC.SAMPLE_DATA_TBL
VALUES ('KS','234-45-6477',27,'4053 0495 0394 0494'),
('TX','234-85-6477',67,'4653 0495 0394 0494'),
('TX','235-45-6477',44,'4053 0755 0394 0494'),
('MD','234-85-6477',81,'4873 0495 0394 4094'),
(['CA','234-85-0877',18,'4653 0495 0084 0494']);

```

Next we need to create a mapping table to reference

```

-- Create a mapping table for row-level access

CREATE OR REPLACE TABLE SNOWTEST.PUBLIC.MAPPING (
    ROLE_ENTITLED varchar, STATE varchar
);

-- Insert mapping values

INSERT INTO SNOWTEST.PUBLIC.MAPPING VALUES ('ROLE1','TX'),
('ROLE1','KS'),
('ROLE1','MD'),
('ROLE1','CA'),
('ROLE1','TX'),
(['ROLE1','MA']),
('ROLE3','TX'),
('ROLE3','KS');

```

Now we will create a row access policy that will enable use the ability to use the mapping to determine what rows you can see.

```

-- Create row access policy

CREATE ROW ACCESS POLICY SNOWTEST.PUBLIC.TEST_POLICY AS
(state_filter varchar) RETURNS BOOLEAN -> -- This will provide a true or false on the row based on the mapping table
CURRENT_ROLE() = 'SUPERADMIN'
OR EXISTS (
    SELECT 1 FROM SNOWTEST.PUBLIC.MAPPING
    WHERE STATE = state_filter
    AND ROLE_ENTITLED = CURRENT_ROLE());

-- Apply the row access policy on the created table

ALTER TABLE SNOWTEST.PUBLIC.SAMPLE_DATA_TBL ADD ROW ACCESS POLICY SNOWTEST.PUBLIC.TEST_POLICY ON (STATE);

```

Let's try different role:

Role 1

	STATE	SSN	AGE	CC
1	KS	234-45-6477	27	4053 0495 0394 0494
2	TX	234-85-6477	67	4653 0495 0394 0494
3	TX	235-45-6477	44	4053 0755 0394 0494
4	MD	234-85-6477	81	4873 0495 0394 4094
5	CA	234-85-0877	18	4653 0495 0084 0494

Role 2

	STATE	SSN	AGE	CC
Query produced no results				

Role 3

	STATE	SSN	AGE	CC
1	KS	234-45-6477	27	4053 0495 0394 0494
2	TX	234-85-6477	67	4653 0495 0394 0494
3	TX	235-45-6477	44	4053 0755 0394 0494

Superadmin

	STATE	SSN	AGE	CC
1	KS	234-45-6477	27	4053 0495 0394 0494
2	TX	234-85-6477	67	4653 0495 0394 0494
3	TX	235-45-6477	44	4053 0755 0394 0494
4	MD	234-85-6477	81	4873 0495 0394 4094
5	CA	234-85-0877	18	4653 0495 0084 0494

Every role can only see the records that meet the mapping requirements. Since Role 2 has no value, nothing is will be visible. Superadmin can see everything.

Test your skills

-- Using row access policies and masking, make it such that the password is masked from everyone, Team A can only see regions 'N' and 'S', Team B can see everything, Team C can only see the region W and has everything by ID masked and TeamD can only see dogs