International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 09 Issue: 08 | Aug 2022 www.irjet.net p-ISSN: 2395-0072

Dynamic Data Masking Mechanism on Cloud Platform

Chirag Dave, Deepa Dave

Chirag Dave, Program Manager, Department of Education
Deepa Dave, Snowflake Lead Consultant, Under Armor

Abstract - *In recent years, Cloud Computing is gaining popularity and has changed the overall business computing environment. It is flexible and cost effective. However, this change has brought its own set of Data Security challenges.*

*In this paper, the author has focused on explaining the application of Dynamic Data Masking policies using ELT tool named Data Build Tool (**DBT**). This paper attempts to describe a detailed approach to create the DDM policies on Cloud Datawarehouse **(Snowflake)** and applying those DDM policies to **DBT models** which will selectively mask the plain text data and view columns at query time within Snowflake before sharing it outside the organization. This white paper addresses the Data masking aspects for most of the projects which use Snowflake and DBT as their Data Warehouse and ELT tool respectively.*

***Key Words*: Cloud Computing, Data Security, Data Sharing, Data Masking, Data Build Tool, Snowflake, DBT Macro, DDM Policies**

1. INTRODUCTION

With humongous Data Sharing capabilities on Cloud Platform, Organization's sensitive information has been the utmost threat, as most of third-party infrastructure can access this information remotely and from anywhere around the world. One of the instrumental solutions to overcome this Data Security threat, is to protect this sensitive information from unauthorized access is with the implementation of **Dynamic Data Masking (DDM)** on Cloud Data Warehouse.

2. BACKGROUND OF THE PROJECT

There are numerous projects that have sensitive Customer and Personally Identifiable Information, which requires special Authentication for anyone in the organization to access it. The Customer may also be responsible to not share this information with Third Party vendors and Consultants. Below outlined is the high-level architecture of the project and the highlighted area is where the masking policies has been applied.

3. SOLUTION IMPLEMENTATION

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.

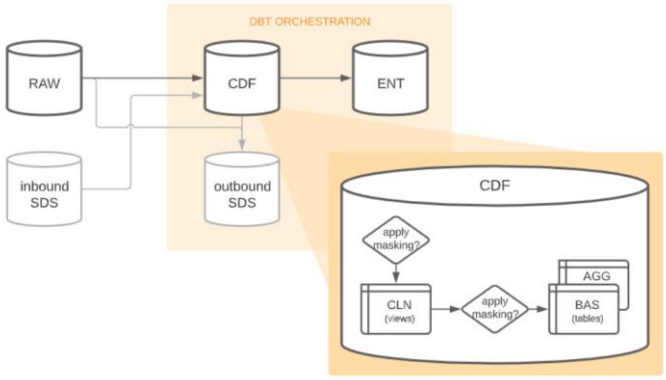

Chart -1: DBT Orchestration

Detailed Description of DDM policies that are available in DBT:

If your role access is approved in the DDM policy you will see the raw value, otherwise a masked value will be returned.

1. **Hashed DDM (for string data types)**

Purpose: Maintains privacy while allowing the column value to still be joinable to other objects that have the same Hashed DDM policy

Policy: hash_mask

Masked Value: sha2(concat(lower(val), 'salt'))

2. **Asterisks DDM (for string data types)**

Purpose: Maintains privacy while allowing approved Snowflake roles to see the raw value.

Policy: asterisks_mask

Masked Value: **********

3. **Asterisks DDM (for binary data types)**

Purpose: Maintains privacy while allowing approved Snowflake roles to see the raw value.

Policy: asterisks_binary_mask

Masked Value: to_binary('**********', 'utf-8')

STEPS TO CREATE MASKING POLICIES IN SNOWFLAKE:

Based on business requirement and the way customer wants to encrypt sensitive information can be implemented using below steps in Snowflake. Customer can choose one of the DDM policies outlined above and mask the data accordingly.

3.1. Masking policies are currently stored in CDF's UTIL schemas and managed by the ddm_masking_admin role.

These policies should be created in each DEV/QA/PROD environment under CDF database.

```
USE role ddm_masking_admin;

// This example creates a "general_mask" policy that returns a SHA256 hash value
// when current/secondary role is not DAPL_ENGINEER_GRP
CREATE OR REPLACE masking policy dapl_cdf_xxx_db.util.general_mask AS  (val string) RETURNS string ->
    CASE
      WHEN is_role_in_session('CDF_ENGINEER_GRP') THEN val
      ELSE sha2(LOWER(val))
    END
;

// This example alters the above policy by returning asterisks instead of SHA256
ALTER masking policy dapl_cdf_xxx_db.util.general_mask set body ->
    CASE
      WHEN is_role_in_session('CDF_ENGINEER_GRP') THEN val
      ELSE '*****'
    END
;
```

Fig -1: Masking Policy Creation

3.2. Once masking policies are created, we would need to grant role access to these policies, and these can be carried using admin role.

```
use role securityadmin;

// This example GRANTs roles TO apply masking policies
GRANT apply ON masking policy dapl_cdf_xxx_db.util.general_mask TO role cdf_engineer_grp;
GRANT apply ON masking policy dapl_cdf_xxx_db.util.general_mask TO role elt_developer_grp;
GRANT apply ON masking policy dapl_cdf_xxx_db.util.general_mask TO role elt_developer_external_grp;
GRANT apply ON masking policy dapl_cdf_xxx_db.util.general_mask TO role elt_dbt_dev_service_grp;
GRANT apply ON masking policy dapl_cdf_xxx_db.util.general_mask TO role elt_dbt_qas_service_grp;
GRANT apply ON masking policy dapl_cdf_xxx_db.util.general_mask TO role elt_dbt_prd_service_grp;

// This example GRANTs roles TO usage ON the UTIL schema
GRANT USAGE ON SCHEMA dapl_cdf_xxx_db.util TO role cdf_engineer_grp;
GRANT USAGE ON SCHEMA dapl_cdf_xxx_db.util TO role elt_developer_grp;
GRANT USAGE ON SCHEMA dapl_cdf_xxx_db.util TO role elt_developer_external_grp;
GRANT USAGE ON SCHEMA dapl_cdf_xxx_db.util TO role elt_dbt_dev_service_grp;
GRANT USAGE ON SCHEMA dapl_cdf_xxx_db.util TO role elt_dbt_qas_service_grp;
GRANT USAGE ON SCHEMA dapl_cdf_xxx_db.util TO role elt_dbt_prd_service_grp;
```

Fig -2: Access Permission

3.3. Writing a DBT post hook macro

We can write a DBT post-hook macro which will be used to apply DDM policies to the models.

```
{% macro apply_ddm_policies_to_model(policies) %}
    {% if policies|length > 0 %}

        {% set model = this %}
        {% set materialization = 'table' if this.materialized == 'table' else 'view' %}

        {% set ns = namespace(sql='') %}
        {% for key, value in policies.items() %}
            {% set ns.sql = ns.sql ~ 'alter ' ~ materialization ~ ' ' ~ model ~ ' modify column ' ~ key ~ ' unset masking policy; ' %}
            {% set ns.sql = ns.sql ~ 'alter ' ~ materialization ~ ' ' ~ model ~ ' modify column ' ~ key ~ ' set masking policy ' ~ this.database ~ '.util.' ~ value ~ '; ' %}
        {% endfor %}

        {% do log('Applying DDM policies:', info=True) %}
        {% do log(ns.sql, info=True) %}
        {% do run_query(ns.sql) %}
        {% do log('Policies applied.', info=True) %}

    {% endif %}
{% endmacro %}
```

Fig -3: Post-hook DBT Macro

3.4. Applying the DDM policies on Individual DBT model

Masking policies can be applied manually; however, it is better to add the configuration in the individual DBT model for easy maintenance of the code which can be re-used anywhere.

1. Manual process:

```
use role cdf_engineer_grp;

// This example applies an existing masking policy to a table column
ALTER TABLE dapl_raw_dev_db.sfmc.subscribers
  MODIFY COLUMN email_address set masking policy dapl_cdf_xxx_db.util.general_mask
;

// This example removes all masking policies from a table column
ALTER TABLE dapl_raw_dev_db.sfmc.subscribers
  MODIFY COLUMN email_address unset masking policy
;
```

Fig -4: Manual Application of DDM Policy

2. Masking configuration embedded inside DBT Model:

```
{{ config(
    alias='bounces',
    post_hook='{{ apply_ddm_policies_to_model(policies={
        "subscriber_key": "hash_mask",
        "email_address": "hash_mask"
    }) }}'
) }}
```

Fig -5: DBT Model Configuration of DDM Policy

This is how data looks like after the DDM policies implementation.

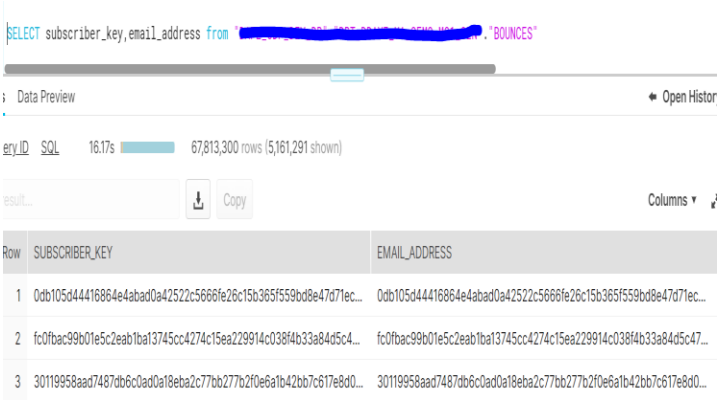

Fig -6: Masking Implementation on Table

We can also apply DDM Policies to All DBT Models in a specific Folder by calling the post-hook macro created in Step 2. Folder level implementation can be done in dbt_project.yml which is the DBT configuration file.

The policies parameter is set to apply the hash_mask policy to column subscriber_key and asterisks_mask policy to column email_address.

Note: The columns subscriber_key and email_address should exist in ALL models.

```
models:
    dapl_core_data_foundation:
        clean:
            ua_sfmc:
                +database: DAPL_CDF
                +schema: DB_xxxx_CLN
                +tags: "dapl"
                +post-hook: '{{ apply_ddm_policies_to_model(policies={"subscriber_key": "general_mask", "email_address": "email_mask"}) }}'
```

Fig -6: dbt_project.yml configuration

3.5. Manage the DDM policies

```
// This describes the policy
describe masking policy dapl_cdf_xxx_db.util.general_mask;

// This shows all objects using the specified masking policy
use database dapl_cdf_xxx_db;
use schema information_schema;
select * from table(information_schema.policy_references(policy_name => 'dapl_cdf_xxx_db.util.general_mask'));
```

Once the DDM policies are implemented in DBT Models, we can easily outsource the data by creating Snowflake's Outbound secure data share.

4. CONCLUSION

Data masking is essential to organizations in adhering and implementing various compliances. Some of the compliance policies include HIPAA, Legal Compliances, IT and Data Sharing Compliances, Financial and Intellectual Property Compliances, etc.

Data Masking is one of the primary methods of storing or transforming the data, while making it inaccessible to non-authorized users within or outside of Organization.

Benefits of the approach mentioned in this white paper are:

- Non-availability of Raw Data in case of Data Loss
- Data protection from Insider hazard within Organization
- Secured data transfer in case of Insecure interfaces for data in motion
- Minimal maintenance of the DBT code
- Reusability of the code is unique
- Masked Data can be used for Joins and Transformations (under special conditions)

REFERENCES

[1] Snowflake Community Documentation.

[2] DBT Cloud Community

[3] Slack Community