OTI Monitoring Object Descriptions

Purpose:

This document contains definitions and explanations of several Snowflake stored procedures and a scheduled task that are utilized in archiving, curating and sharing Snowflake metadata for use in centralized monitoring of multiple accounts within a single organization.

Note: Stored procedure explanations generated using Snowflake Cortex Complete Function with the mixtral-8x7b LLM as described in this article:

- https://medium.com/snowflake/snowflake-cortex-driven-toolbox-05bdfaa2a56f

Source Account Objects:

PROCEDURE: SNOWFLAKE_ARCHIVE_LOAD_DATA

General Explanation: The code is a stored procedure that creates or replaces views in a Snowflake database based on the metadata stored in the METADATA_ARCHIVE_CONTROL_INFO table. The stored procedure loops through the table, generates the CREATE VIEW statements, and then executes them. It also logs any errors that occur during the execution of the CREATE VIEW statements.

Breakdown of Steps:

The stored procedure begins by declaring variables and creating a cursor for the METADATA_ARCHIVE_CONTROL_INFO table.

It then creates a temporary table called sqlstmt to store the CREATE VIEW statements that will be generated.

The stored procedure loops through the METADATA_ARCHIVE_CONTROL_INFO table using the table_cursor cursor.

For each row in the table, it generates a CREATE VIEW statement using the DESCRIBE VIEW command and the listagg function.

The generated CREATE VIEW statement is then inserted into the sqlstmt table.

The stored procedure then loops through the sqlstmt table and executes each CREATE VIEW statement using the EXECUTE IMMEDIATE command.

If an error occurs during the execution of a CREATE VIEW statement, the error is logged in the error_array array.

After all the CREATE VIEW statements have been executed, the stored procedure returns the msg_array array, which contains messages indicating whether each CREATE VIEW statement was successful.

Output: The output of the stored procedure is an array of messages indicating whether each CREATE VIEW statement was successful. The messages are stored in the msg_array array and are returned by the stored procedure.

Here is an example of what the output might look like:

```
[
  "View created  ::  CREATE OR REPLACE VIEW myschema.myview COPY GRANTS as select PAYLOAD:col1::string as col1, PAYLOAD:col2::number as col2 from CONTROLS.myview",
  "View created  ::  CREATE OR REPLACE VIEW myschema.myotherview COPY GRANTS as select PAYLOAD:col3::string as col3, PAYLOAD:col4::date as col4 from CONTROLS.myotherview"
]
```

If there were any errors during the execution of the CREATE VIEW statements, they would be logged in the error_array array and could be retrieved separately.

PROCEDURE: SNOWFLAKE_ARCHIVE_CREATE_VIEWS()

General Explanation

This Stored Procedure performs the following tasks:

It declares variables, arrays, and a cursor.

It creates a temporary table sqlstmt to store DDL statements.

It loops through the METADATA_ARCHIVE_CONTROL_INFO table and creates CREATE VIEW DDL statements.

It loops through the sqlstmt table and executes the SQL statements.

It logs any errors that occur during the execution of the SQL statements.

It returns an array of messages about the executed SQL statements.

Breakdown of Steps

Declarations:

Variables: t, db, s, fqn, output are declared as varchar.

Arrays: msg_array and error_array are declared as arrays of strings (array).

Cursor: table_cursor is declared to fetch data from the METADATA_ARCHIVE_CONTROL_INFO table.

Create a temporary table:

sqlstmt is created as a temporary table with a single column ddl to store DDL statements.

Loop through the METADATA_ARCHIVE_CONTROL_INFO table:

The cursor table_cursor is used to fetch data from the METADATA_ARCHIVE_CONTROL_INFO table.

For each row fetched, it constructs a CREATE VIEW DDL statement and inserts it into the sqlstmt table.

Loop through the sqlstmt table and execute SQL:

A new cursor cur is declared to fetch data from the sqlstmt table.

For each row fetched, it gets the DDL statement, executes it, and logs any errors that occur.

Log errors:

If any errors occur during the execution of the SQL statements, they are logged in the error_array array.

Return an array of messages:

The msg_array array, which contains messages about the executed SQL statements, is returned.

Output

The output of the stored procedure is the msg_array array, which contains messages about the executed SQL statements. The messages are in the format View created :: <DDL statement>. If any errors occur, they are logged in the error_array array. The format of the error messages is ERROR creating view for <FQN>. Error Message: <Error message> -- DDL: <DDL statement>.

PROCEDURE: SHARE_ALL_OBJECTS

General Explanation

This is a Snowflake Stored Procedure written in SQL that creates a share for a specified database and schema, allowing non-secure objects to be shared. It then generates and executes a series of grant statements to share the objects in the specified schema with the created share.

The stored procedure follows these steps:

Create the outbound share with the specified name if it doesn't already exist.

Allow sharing of non-secure objects in the share.

Create a temporary table to store the grant statements.

Insert grant statements for the database, schema, tables, external tables, iceberg tables, and materialized views.

Log grant statements for functions and views in the database.

Execute the grant statements and store the results (success or error messages) in arrays.

Return the arrays containing the results.

Breakdown of Steps

Step 1: Create the outbound share

The `CREATE SHARE` command creates a new outbound share with the specified name if it doesn't already exist.

Step 2: Allow sharing of non-secure objects

The `ALTER SHARE` command sets the `SECURE_OBJECTS_ONLY` parameter to `FALSE`, allowing non-secure objects to be shared.

Step 3: Create a temporary table for grant statements

A temporary table named `ADMIN_UTLITIES.ADMIN_MONITORING.GRANT_STATEMENTS` is created to store the grant statements.

Step 4: Insert grant statements for database, schema, tables, etc.

Grant statements are inserted into the temporary table for the database, schema, tables, external tables, iceberg tables, and materialized views.

Step 5: Log grant statements for functions and views

The `SHOW FUNCTIONS` command is used to retrieve a list of functions in the database. Grant statements are then inserted into the temporary table for each function.

The `INFORMATION_SCHEMA.VIEWS` table is queried to retrieve a list of views in the specified schema. Grant statements are inserted into the temporary table for each view.

Step 6: Execute the grant statements

A cursor is used to iterate through the grant statements in the temporary table. Each grant statement is executed using the EXECUTE IMMEDIATE command.

Success or error messages are stored in arrays based on the execution result.

Step 7: Return the arrays containing the results

The arrays containing the success or error messages are concatenated and returned as the output of the stored procedure.

Output

The output of the stored procedure is an array containing the success or error messages for each grant statement executed. The array can be accessed and processed using Snowflake SQL commands or a Snowflake client.

PROCEDURE : SNOWFLAKE_ARCHIVE_ELEVATED_ROLE_USERS

General Explanation

This code creates a stored procedure in Snowflake that collects data about users with elevated roles. The procedure creates two tables, `ELEVATED_ROLE_USERS` and `ROLE_HIERARCHIES`, and populates them with data about users and roles with elevated privileges.

The `ELEVATED_ROLE_USERS` table contains information about users who have been granted elevated roles or privileges. The `ROLE_HIERARCHIES` table contains information about the hierarchy of roles and the privileges granted to each role.

The procedure uses recursive queries to build the hierarchy of roles and privileges. It then inserts the data into the `ROLE_HIERARCHIES` table and joins it with data from the `SNOWFLAKE.ACCOUNT_USAGE.GRANTS_TO_USERS` view to populate the `ELEVATED_ROLE_USERS` table.

Breakdown of Steps

1. Create the `ELEVATED_ROLE_USERS` and `ROLE_HIERARCHIES` tables if they do not already exist.
2. Build the hierarchy of roles and privileges using recursive queries.
3. Insert the data into the `ROLE_HIERARCHIES` table.
4. Join the `ROLE_HIERARCHIES` table with data from the `SNOWFLAKE.ACCOUNT_USAGE.GRANTS_TO_USERS` view to populate the `ELEVATED_ROLE_USERS` table.

Output

The output of this procedure is the creation of the `ELEVATED_ROLE_USERS` and `ROLE_HIERARCHIES` tables, which contain information about users and roles with elevated privileges. The `ELEVATED_ROLE_USERS` table contains the following columns:

- `USER_NAME`: the name of the user
- `ELEVATED_ROLE`: the name of the elevated role granted to the user (NULL if the user has a privilege but no elevated role)
- `GRANTED_VIA_ROLE`: the name of the role that granted the elevated role or privilege to the user
- `GRANT_PATH`: the hierarchy of roles and privileges that led to the granting of the elevated role or privilege
- `CREATED_ON`: the timestamp when the user was granted the elevated role or privilege
- `ELEVATED_PRIVILEGE`: the name of the privilege granted to the user (NULL if the user has an elevated role but no privilege)
- `SNAPSHOT_TIMESTAMP`: the timestamp when the data was collected

The `ROLE_HIERARCHIES` table contains the following columns:

- `ELEVATED_ROLE`: the name of the elevated role
- `ROLE_PATH`: the hierarchy of roles and privileges that led to the elevated role
- `PATH_ARRAY`: the hierarchy of roles and privileges as an array
- `ROLE_NAME`: the name of each role in the hierarchy
- `PRIVILEGE`: the name of the privilege granted to the role (NULL if the role has no privilege)
- `SNAPSHOT_TIMESTAMP`: the timestamp when the data was collected

TASK: TASK_LOAD_SNOWFLAKE_ARCHIVE_DATA

General Explanation

The code creates a new task called `task_load_snowflake_archive_data` that runs on a cron schedule, using the specified timezone and warehouse. The task consists of three calls to stored procedures, which will be executed in order when the task runs.

Breakdown of Steps

1. CREATE OR REPLACE TASK task_load_snowflake_archive_data: This line creates or replaces the task named `task_load_snowflake_archive_data`.
2. SCHEDULE = 'USING CRON 0 0 * * * America/Chicago': This line sets the schedule for the task using the CRON syntax. The task will run every day at midnight, Central Time (America/Chicago timezone).
3. WAREHOUSE = 'tasty_bi_wh': This line specifies the Snowflake warehouse to be used for executing the task. The warehouse named 'tasty_bi_wh' will be used.
4. AS BEGIN: This line starts the task definition.
5. CALL SNOWFLAKE_ARCHIVE_LOAD_DATA();: This line calls the stored procedure `SNOWFLAKE_ARCHIVE_LOAD_DATA`. This procedure is responsible for loading data from the Snowflake archive.
6. CALL SNOWFLAKE_ARCHIVE_CREATE_VIEWS();: This line calls the stored procedure `SNOWFLAKE_ARCHIVE_CREATE_VIEWS`. This procedure is responsible for creating views based on the data in the Snowflake archive.
7. CALL SNOWFLAKE_ARCHIVE_ELEVATED_ROLE_USERS();: This line calls the stored procedure `SNOWFLAKE_ARCHIVE_ELEVATED_ROLE_USERS`. This procedure is responsible for managing users with elevated roles related to the Snowflake archive.
8. END;: This line ends the task definition.

Output

The output of this code is the creation of the task `task_load_snowflake_archive_data`, which will run automatically based on the specified CRON schedule. The task will execute the three stored procedures in order when it runs. The output will not be visible in the Snowflake worksheet, but you can check the task status and history in the Snowflake web UI or by querying the `TASK_HISTORY` and `TASK` tables.

TASK: TASK_LOAD_ACCOUNT_PARAMETERS

General Explanation

This code creates or replaces a task named `task_load_account_parameters` in Snowflake. This task is scheduled to run every day at midnight Central Time (Chicago timezone). It executes a block of SQL code in the `tasty_bi_wh` warehouse.

The task performs the following actions:

1. It shows the current account parameters.

2. It inserts the result of the `SHOW PARAMETERS IN ACCOUNT;` command into the `CONTROLS.ACCOUNT_PARAMATERS` table.

Breakdown of Steps

Step 1: Task Creation

```
CREATE OR REPLACE TASK task_load_account_parameters
```

This line creates or replaces the task named `task_load_account_parameters`.

Step 2: Schedule the Task

```
SCHEDULE = 'USING CRON 0 0 * * * America/Chicago'
```

This line sets the schedule for the task to run every day at midnight Central Time (Chicago timezone).

Step 3: Specify the Warehouse

```
WAREHOUSE = 'tasty_bi_wh' --CHANGE WAREHOUSE
```

This line specifies the warehouse where the SQL code will be executed. In this case, it's the `tasty_bi_wh` warehouse.

Step 4: SQL Code Block

```
BEGIN ... END;
```

This block of SQL code will be executed when the task runs.

Step 5: Show Account Parameters

```
SHOW PARAMETERS IN ACCOUNT;
```

This command shows the current account parameters.

Step 6: Insert Result into Table

```
INSERT OVERWRITE INTO CONTROLS.ACCOUNT_PARAMATERS

(KEY, VALUE,DEFAULT,LEVEL,DESCRIPTION, TYPE)

SELECT

      "key" AS KEY,

      "value" AS VALUE,

      "default" AS DEFAULT,

      "level" AS LEVEL,

      "description" AS DESCRIPTION,
      "type" AS TYPE

      FROM TABLE(RESULT_SCAN(LAST_QUERY_ID()));
```

This command inserts the result of the `SHOW PARAMETERS IN ACCOUNT;` command into the `CONTROLS.ACCOUNT_PARAMATERS` table. It maps the columns of the result to the columns of the table.

Output

The output of this code will not be visible in the console, as it creates a task that runs in the background. The task will insert the account parameters into the `CONTROLS.ACCOUNT_PARAMATERS` table every day at midnight Central Time (Chicago timezone). You can check the content of the `CONTROLS.ACCOUNT_PARAMATERS` table to see the output.

Target Account Objects:

PROCEDURE : UNION_BUILDER

General Explanation

This stored procedure, `UNION_BUILDER`, is designed to create or replace views in a Snowflake database. It takes a `DB_PREFIX` parameter, which is used to filter the tables and views to be included in the views created by the procedure.

The procedure creates a temporary table called `account_shares` that contains the names of all databases in the account that match the `DB_PREFIX` pattern. It then loops through the names of the objects in `account_shares` and creates a view for each one. The view is created as a union of all the tables and views in the database that have the same schema name as the object name. If there are any errors while creating the views, they are logged in an error array.

Breakdown of Steps

1. The procedure declares several variables, including a cursor for the object list, a variable to hold the name of each object, and two arrays to hold the names of the objects and any errors that occur while creating the views.

2. The procedure shows the databases in the account and creates a temporary table called `account_shares` that contains the names of all databases that match the `DB_PREFIX` pattern.
3. The procedure sets the value of the `db` variable to the name of the first database in `account_shares`.
4. The procedure shows the tables and views in the `db` database and creates a temporary table called `object_list` that contains the names of all tables and views in the database that have a schema name that is not `INFORMATION_SCHEMA`.
5. The procedure loops through the names of the objects in `object_list` and creates a view for each one.
6. For each object, the procedure splits the name into the schema name and the object name, and constructs a dynamic SQL statement to create a view that is a union of all the tables and views in the database that have the same schema name as the object name.
7. The procedure executes the dynamic SQL statement to create the view.
8. If there are any errors while creating the view, the procedure logs the error message in the `error_array` array.
9. After all the views have been created, the procedure returns the `error_array` array.

Output

The output of the procedure is an array of error messages, if any errors occurred while creating the views. If no errors occurred, the array will be empty. The array can be accessed by calling the `UNION_BUILDER` procedure with a `DB_PREFIX` parameter and using the `RETURN` statement to retrieve the array.

For example, if the `UNION_BUILDER` procedure is called with a `DB_PREFIX` parameter of `mydb`, the output might look something like this:

```
[]
```

This indicates that no errors occurred while creating the views. If there were errors, the output might look something like this:

```
[

  "ERROR creating view for mydb.table1.view1. Error Message: SQL compilation error: Object 'mydb.table1.view1' does not exist or not authorized.",

  "ERROR creating view for mydb.table2.view2. Error Message: Object 'mydb.table2.view2' does not exist or not authorized."
]
```
This indicates that there were two errors while creating the views, and the error messages are included in the `error_array` array.

ALERT: TC_CRITICAL_FINDINGS_ALERT

General Explanation
This code creates two alerts in Snowflake, a cloud-based data warehousing platform. These alerts are designed to notify the user when certain conditions are met. The first alert, `TC_CRITICAL_FINDINGS_ALERT`, is triggered when there are critical findings in the trust center. The second alert, `ELEVATED_PRIVILEGES_ALERT`, is triggered when there are users granted elevated privileges or roles within the system.

Code Breakdown
1. TC_CRITICAL_FINDINGS_ALERT
Step 1: Alert Definition

- Define the alert named `TC_CRITICAL_FINDINGS_ALERT` using the `CREATE OR REPLACE ALERT` statement.
- Set the alert to run every 1440 minutes (once per day).
- Use the `$WH_NAME` variable to specify the warehouse to use for the alert.

Step 2: Alert Condition
- The alert is triggered when a critical finding exists in the `FINDINGS` table.
- The `EXISTS` keyword checks if any records meet the specified condition.
- The `QUALIFY` clause filters the results to only show the most recent finding for each `ACCOUNT_DATABASE` and `SCANNER_ID` pair.

Step 3: Alert Action
- When the alert is triggered, it inserts a record into the `alert_event_log` table.
- The `WITH` clause creates a CTE (Common Table Expression) named `CRITICAL_FINDINGS` to store the relevant critical findings.
- The `SELECT` statement retrieves the necessary information from the `CRITICAL_FINDINGS` CTE and inserts it into the `alert_event_log` table.

2. ELEVATED_PRIVILEGES_ALERT
Step 1: Alert Definition
- Define the alert named `ELEVATED_PRIVILEGES_ALERT` using the `CREATE OR REPLACE ALERT` statement.
- Set the alert to run every 1440 minutes (once per day).
- Use the `$WH_NAME` variable to specify the warehouse to use for the alert.

Step 2: Alert Condition

- The alert is triggered when a user is granted elevated privileges or roles within the system.
- The `EXISTS` keyword checks if any records meet the specified condition.
- The `COALESCE` function is used to provide a default value for `LAST_SUCCESSFUL_SCHEDULED_TIME()` if it is null.

Step 3: Alert Action

- When the alert is triggered, it inserts a record into the `alert_event_log` table.
- The `SELECT` statement retrieves the necessary information from the `ELEVATED_ROLE_USERS` table and inserts it into the `alert_event_log` table.

Output

The output of this code is the creation of two alerts, `TC_CRITICAL_FINDINGS_ALERT` and `ELEVATED_PRIVILEGES_ALERT`, which will monitor the Snowflake system for critical trust center findings and elevated privileges, respectively. When triggered, these alerts will insert records into the `alert_event_log` table, which can be used to track and respond to these events.

ALERT: ACCOUNT_PARAMETERS_ALERT

General Explanation

This code creates or replaces an alert in Snowflake called `ACCOUNT_PARAMETERS_ALERT`. This alert checks for specific account parameters and their values in the `ORG_MONITORING_UTILITIES.CENTRALIZED_DATA.ACCOUNT_PARAMETERS` table and triggers an alert if the conditions are met.

The alert is scheduled to run every 1440 minutes (once a day). It checks for the following account parameters:

1. `REQUIRE_STORAGE_INTEGRATION_FOR_STAGE_CREATION` with value 'false'
2. `REQUIRE_STORAGE_INTEGRATION_FOR_STAGE_OPERATION` with value 'false'
3. `PREVENT_UNLOAD_TO_INLINE_URL` with value 'false'
4. `PRE_SIGNED_URL` with value 'true'

If any of these parameters have the specified values and their `COLLECTION_DATE` is later than the last successful scheduled time minus one day or January 1, 2025, the alert is triggered.

When the alert is triggered, it inserts a row into the `alert_event_log` table with the current scheduled time, the account database, the event type (`ACCOUNT_PARAMETERS_ALERT`), and a message describing the account parameter that was found to be out of compliance.

Breakdown of Steps

1. Create or replace the `ACCOUNT_PARAMETERS_ALERT` with the specified warehouse, schedule, and conditions.
2. Check if any of the specified account parameters have the specified values and their `COLLECTION_DATE` is later than the last successful scheduled time minus one day or January 1, 2025.
3. If the conditions are met, insert a row into the `alert_event_log` table with the current scheduled time, the account database, the event type (`ACCOUNT_PARAMETERS_ALERT`), and a message describing the account parameter that was found to be out of compliance.

Output

The output of this code is the creation or replacement of the `ACCOUNT_PARAMETERS_ALERT` and the insertion of rows into the `alert_event_log` table when the alert is triggered. The `alert_event_log` table will contain the following columns:

- `event_timestamp`: The time the alert was triggered.
- `event_database`: The account database where the account parameter was found to be out of compliance.
- `event_type`: The type of alert that was triggered (`ACCOUNT_PARAMETERS_ALERT`).
- `event_message`: A message describing the account parameter that was found to be out of compliance.

ALERT: ELEVATED_PRIVILEGES_ALERT

General Explanation:

The code creates an alert in Snowflake that checks for any new elevated privileges granted to users in the past day. If any new elevated privileges are found, an alert is triggered and an entry is added to the `alert_event_log` table.

Breakdown of Steps:

1. The `CREATE OR REPLACE ALERT` statement creates or replaces an alert named `ELEVATED_PRIVILEGES_ALERT` in the Snowflake database.
2. The `WAREHOUSE` parameter specifies the name of the Snowflake warehouse to use for running the alert.
3. The `SCHEDULE` parameter sets the alert to run every 1440 minutes (once per day).

4. The `IF` statement checks if there are any new elevated privileges granted to users in the past day.
5. If new elevated privileges are found, the `THEN` statement inserts an entry into the `alert_event_log` table with the following information:
    - `event_timestamp`: the time the alert was triggered
    - `event_database`: the name of the database where the elevated privilege was granted
    - `event_type`: the type of alert (`ELEVATED_PRIVILEGES_ALERT`)
    - `event_message`: a message describing the elevated privilege that was granted, including the user who received the privilege, the type of privilege or role granted, and the path through which the privilege was granted.

Output:

The output of this code is the creation of the `ELEVATED_PRIVILEGES_ALERT` alert in the Snowflake database. If the alert is triggered, an entry will be added to the `alert_event_log` table with information about the elevated privilege that was granted. The specific output will depend on the data in the `ORG_MONITORING_UTILITIES.CENTRALIZED_DATA.ELEVATED_ROLE_USERS` table and the configuration of the `alert_event_log` table.

ALERT: alert_email_events_hourly

General Explanation

This code creates an alert in Snowflake that triggers every hour. When triggered, it sends an email with information about specific event types that have occurred since the last successful scheduled time. The email is sent using a notification integration and recipients specified in the `event_recipients` table.

Breakdown of Steps

1. Alert Creation: The alert is named `alert_email_events_hourly` and is scheduled to run every 60 minutes.
2. Check for Existing Alert Events: The alert checks if there are any existing alert events between the last successful scheduled time and the current scheduled time. If there are, the alert proceeds to the next step.
3. Prepare Data for Email: The code then prepares the data for the email by:
    - Creating a Common Table Expression (CTE) named `cte_event_recipients` that groups event types and their corresponding notification integration and recipients' email addresses.

    - Creating a CTE named `cte_events` that groups event types and their corresponding event messages.
    - Joining `cte_events` with `cte_event_recipients` to get the email subject, notification integration, recipients' email addresses, alert email ID, email body, and email events.
4. Send Email: For each record in the result set, the code:
    - Constructs a SQL statement to call the `SYSTEM$SEND_EMAIL` function.
    - Executes the SQL statement to send the email.
    - Inserts a record into the `ALERT_EMAIL_LOG` table with the email details and the query ID.
5. Handle Errors: If there is a statement error, the code:
    - Gets the SQL error message and code.
    - Inserts a record into the `ALERT_EMAIL_LOG` table with the email details, query ID, error code, and error message.

Output

The output of this code is an email sent to the specified recipients with information about the specified event types that have occurred since the last successful scheduled time. The email includes the event type, event message, and event ID. If there is an error, the error is logged in the `ALERT_EMAIL_LOG` table.

ALERT: alert_email_events_daily

General Explanation

This code creates an alert in Snowflake that triggers every hour. When triggered, it sends an email with information about specific event types that have occurred since the last successful scheduled time. The email is sent using a notification integration and recipients specified in the `event_recipients` table.

Breakdown of Steps

6. Alert Creation: The alert is named `alert_email_events_hourly` and is scheduled to run every 1440 minutes.
7. Check for Existing Alert Events: The alert checks if there are any existing alert events between the last successful scheduled time and the current scheduled time. If there are, the alert proceeds to the next step.
8. Prepare Data for Email: The code then prepares the data for the email by:
    - Creating a Common Table Expression (CTE) named `cte_event_recipients` that groups event types and their corresponding notification integration and recipients' email addresses.

   - Creating a CTE named `cte_events` that groups event types and their corresponding event messages.
   - Joining `cte_events` with `cte_event_recipients` to get the email subject, notification integration, recipients' email addresses, alert email ID, email body, and email events.
9. Send Email: For each record in the result set, the code:
   - Constructs a SQL statement to call the `SYSTEM$SEND_EMAIL` function.
   - Executes the SQL statement to send the email.
   - Inserts a record into the `ALERT_EMAIL_LOG` table with the email details and the query ID.
10. Handle Errors: If there is a statement error, the code:
   - Gets the SQL error message and code.
   - Inserts a record into the `ALERT_EMAIL_LOG` table with the email details, query ID, error code, and error message.

Output

The output of this code is an email sent to the specified recipients with information about the specified event types that have occurred since the last successful scheduled time. The email includes the event type, event message, and event ID. If there is an error, the error is logged in the `ALERT_EMAIL_LOG` table.

ALERT: alert_failures

General Explanation

The code creates an alert called `alert_failures` in Snowflake. The alert checks for any failures in the last 24 hours and sends an email notification if there are any. The alert is scheduled to run every 1440 minutes (24 hours). The email notification includes the alert name, database name, schema name, state, and error message.

Breakdown of Steps

1. `CREATE OR REPLACE ALERT alert_failures`: This line creates or replaces an alert called `alert_failures`.
2. `WAREHOUSE = $WH_NAME`: This line specifies the warehouse to be used for the alert.
3. `SCHEDULE = '1440 MINUTE'`: This line specifies the schedule for the alert to run every 1440 minutes (24 hours).
4. `IF (EXISTS (...))`: This line checks if there are any failures in the last 24 hours.
5. `SELECT 1 FROM TABLE(INFORMATION_SCHEMA.ALERT_HISTORY(RESULT_LIMIT => 10000)) WHERE state IN ('CONDITION_FAILED','ACTION_FAILED','FAILED') AND`

`SCHEDULED_TIME > SNOWFLAKE.ALERT.LAST_SUCCESSFUL_SCHEDULED_TIME() LIMIT 1`: This line selects the first row from the `ALERT_HISTORY` table where the state is either `CONDITION_FAILED`, `ACTION_FAILED`, or `FAILED` and the scheduled time is after the last successful scheduled time.

6. `THEN`: This line specifies the action to be taken if there are any failures in the last 24 hours.
7. `CALL SYSTEM$SEND_EMAIL('<integration_name>', '<email.address1@mass.gov>,<email.address2@mass.gov>', 'Alert Failures', (WITH CTE AS (SELECT 'ALERT: ' || ifnull(NAME,'') || ' - DB: ' || ifnull(DATABASE_NAME,'') || ' - SCH: ' || ifnull(SCHEMA_NAME,'') || ' - STATE: ' || ifnull(STATE,'') || ' - ERROR: ' || SQL_ERROR_MESSAGE as mail_text FROM TABLE(INFORMATION_SCHEMA.ALERT_HISTORY(RESULT_LIMIT => 10000)) WHERE state IN ('CONDITION_FAILED','ACTION_FAILED','FAILED') AND SCHEDULED_TIME > SNOWFLAKE.ALERT.LAST_SUCCESSFUL_SCHEDULED_TIME()) SELECT listagg(mail_text||'\\n') FROM CTE));`: This line sends an email notification with the alert name, database name, schema name, state, and error message.

Output

The output of this code is the creation or replacement of an alert called `alert_failures` in Snowflake. If there are any failures in the last 24 hours, an email notification will be sent to the specified email addresses with the alert name, database name, schema name, state, and error message.