VCE4Plus

Choose the version that fits your needs	PDF Version	Desktop Test Engine	Online Test Engine
Latest and Up-to-Date exam dumps with real exam questions answers.	✓	✓	✓
Get 12-Months free updates without any extra charges.	✓	✓	✓
Experience same exam environment before appearing in the certification exam.	✗	✓	✓
100% exam passing guarante in the first attempt.	✓	✓	✓
20% discount on more than one license and 30% discount on 5+ license purchases.	✗	✓	✓
100% secure purchase on SSL.	✓	✓	✓
Completely private purchase without sharing your personal info with anyone.	✓	✓	✓

Exam : **DEA-C02**

Title : SnowPro Advanced: Data Engineer (DEA-C02)

Vendor : Snowflake

Version : DEMO

NO.1 You are tasked with loading a large dataset (50TB) of JSON files into Snowflake. The JSON files are complex, deeply nested, and irregularly structured. You want to maximize loading performance while minimizing storage costs and ensuring data integrity. You have a dedicated Snowflake virtual warehouse (X-Large).
Which combination of approaches would be MOST effective?
A. Use Snowpipe with auto-ingest, create a single VARIANT column in your target table, and rely solely on Snowflake's automatic schema detection.
B. Pre-process the JSON data using a Python script with Pandas to flatten the structure and convert it into a relational format like CSV. Then, load the CSV files using the COPY INTO command with gzip compression.
C. Use Snowpipe with auto-ingest, create a raw VARIANT column alongside projected relational columns for frequently accessed fields, and use search optimization on those projected columns.
D. Load the JSON data using the COPY INTO command with no pre-processing. Create a VIEW on top of the raw VARIANT column to flatten the data for querying.
E. Load the JSON data using the COPY INTO command with gzip compression. Create a raw VARIANT column alongside projected relational columns for frequently accessed fields, and use materialized views to improve query performance.
Answer: C
Explanation:
Option C is the most effective. Snowpipe provides continuous loading. A raw VARIANT column captures all data, and projecting commonly accessed fields into relational columns optimizes query performance. Search optimization on the projected columns allows for faster filtering and lookups. Options A, B, D, and E have trade-offs. A lacks optimized querying and can lead to expensive computations on the variant column. B requires pre-processing and may lose data fidelity. D impacts query performance due to runtime flattening. E introduces complexities with materialized view maintenance.

NO.2 You are tasked with migrating data from a legacy SQL Server database to Snowflake. One of the tables, 'ORDERS' , contains a column 'ORDER DETAILS that holds concatenated string data representing multiple order items. The data is formatted as 'iteml :qtyl ;item2:qty2;...'. You need to transform this string data into a JSON array of objects, where each object represents an item with 'name' and 'quantity' fields. Which of the following steps and functions would you use in Snowflake to achieve this transformation, in addition to loading the data?
A. Use to split the string into rows, then use 'SPLIT to separate item name and quantity, and finally use 'OBJECT_CONSTRUCT and to create the JSON array.
B. Use ' to extract item names and quantities, then use 'ARRAY_CONSTRUCT and 'OBJECT_CONSTRUCT to create the JSON array.
C. Use ' STRTOK TO ARRAY' to split the string into an array, then iterate through the array using a JavaScript UDF to create the JSON objects.
D. Use 'SPLIT with ';' as delimiter, then apply 'SPLIT again with ':' as delimiter. Finally, construct the JSON array using 'ARRAY_AGG' and 'OBJECT CONSTRUCT
E. Utilize a Java UDF to parse the string and directly generate the JSON array.
Answer: A,D
Explanation:

Options A and D correctly outline the process. (A) and multiple 'SPLIT calls (D) are valid approaches to break down the concatenated string. Then, 'OBJECT_CONSTRUCT builds the individual JSON objects, and aggregates them into a JSON array. While Javascript or Java UDFs (C, E) could solve the problem, they are generally less efficient than Snowflake's built-in functions. (B) might work but is overkill for this simple splitting task, also you would still need to combine the extracted arrays for items and quantities.

NO.3 You are tasked with sharing a subset of a customer table (CUSTOMER DATA') residing in your organization's Snowflake account with a partner organization. You need to mask personally identifiable information (PII) while providing near real-time updates. You decide to use a secure view. Which of the following SQL statements is the MOST efficient and secure way to accomplish this, assuming the partner only needs 'customer id', 'masked_email', 'city', and 'state'? The email should be masked using SHA256.

○ ```sql CREATE OR REPLACE SECURE VIEW PARTNER_CUSTOMER_VIEW AS SELECT customer_id, SHA2(email, 256) as masked_email, city, state FROM CUSTOMER_DATA; ```

○ ```sql CREATE VIEW PARTNER_CUSTOMER_VIEW AS SELECT customer_id, SHA2(email, 256) as masked_email, city, state FROM CUSTOMER_DATA; GRANT SELECT ON VIEW PARTNER_CUSTOMER_VIEW TO SHARE; ```

○ ```sql CREATE OR REPLACE SECURE VIEW PARTNER_CUSTOMER_VIEW AS SELECT customer_id, SYSTEM$MASK_EMAIL(email) as masked_email, city, state FROM CUSTOMER_DATA; ```

○ ```sql CREATE OR REPLACE VIEW PARTNER_CUSTOMER_VIEW AS SELECT customer_id, SHA2(email, 256) as masked_email, city, state FROM CUSTOMER_DATA; GRANT SELECT ON VIEW PARTNER_CUSTOMER_VIEW TO SHARE; ```

○ ```sql CREATE OR REPLACE SECURE VIEW PARTNER_CUSTOMER_VIEW AS SELECT customer_id, SHA2(email, 256) as masked_email, city, state FROM CUSTOMER_DATA; GRANT SELECT ON VIEW PARTNER_CUSTOMER_VIEW TO SHARE; ```

A. Option A

B. Option B

C. Option C

D. Option D

E. Option E

Answer: A

Explanation:

Option A correctly creates a SECURE VIEW, which is essential for data sharing as it prevents the partner from seeing the underlying table definition. It directly masks the email using SHA256. While Option C uses a hypothetical 'SYSTEM$MASK EMAIL' function, SHA256 is a readily available and standard masking method. Options B, D, and E do not create a secure view or properly grant access for data sharing. The correct method to share is to create secure view and then share using the share object.

NO.4 You're managing a Snowflake data warehouse and need to create a development environment for testing a complex stored procedure that updates a critical table, 'SALES DATA'. The procedure is located in the 'PRODUCTION' database and you want to ensure minimal impact to the production environment during development. You decide to use cloning and time travel. Which of the following strategies is the MOST efficient and safest approach to achieve this, minimizing downtime and resource consumption in production?

A. Clone the entire 'PRODUCTION' database into a new development database. This ensures developers have access to all necessary data and dependencies but consumes significant storage and may impact production performance during the cloning process.

B. Clone only the 'SALES DATA' table into a development database. This minimizes storage

consumption but requires developers to manually recreate or mock any dependencies the stored procedure has on other tables in the 'PRODUCTION' database.

C. Create a snapshot of the 'SALES DATA' table using Time Travel at a specific timestamp (e.g., 1 hour ago), then clone only the stored procedure, updating it to point to the Time Travel version of 'SALES DATA' in the development environment. This provides a consistent dataset for testing while minimizing the impact on production and cloned data volumes.

D. Clone the "PRODUCTION' database. Immediately after cloning, use Time Travel to revert the 'SALES_DATR table in the development database to a state before the stored procedure was last run in production. Then clone the stored procedure itself. This gives a starting point without the procedure's impact.

E. Clone the schema in which 'SALES_DATX is stored along with the stored procedure. Use time travel on the cloned schema to revert all objects in the schema to a point in time before the stored procedure was last run, then update the stored procedure to point to the cloned schema. This gives a consistent starting point for testing in development.

Answer: E

Explanation:

Option E offers the best balance of minimal impact and realistic testing. Cloning the entire database (A) is resource-intensive. Cloning only the table (B) requires significant manual setup to address dependencies. Option C might result in unpredictable behavior if any data dependencies exist that rely on related tables. Option D is almost correct, but the risk is that other objects in the 'PRODUCTION' database schema might change resulting in incomplete testing. Cloning the schema and using Time Travel on the schema level before updating the procedure gives the most consistent and efficient development setup and the best balance.

NO.5 A data engineer is tasked with implementing a data governance strategy in Snowflake. They need to automatically apply a tag 'PII CLASSIFICATION' to all columns containing Personally Identifiable Information (PII). Given the following requirements: 1. The tag must be applied as close to data ingestion as possible. 2. The tagging process should be automated and scalable. 3. The tag value should be dynamically set based on a regular expression match against column names and data types. Which of the following approaches would be MOST effective and efficient in achieving these goals?

A. Create a Snowflake Task that runs daily, querying the INFORMATION SCHEMCOLUMNS view, identifying potential PII columns based on regular expressions, and then executing ALTER TABLE ... ALTER COLUMN ... SET TAG commands.

B. Implement a custom application using the Snowflake JDBC driver to periodically scan table schemas, detect PII columns, and apply tags using dynamic SQL.

C. Use Snowflake's Event Tables in conjunction with a stream and task. Configure the stream to capture DDL changes, and the task to evaluate new columns and apply the tag based on the column metadata using regular expressions.

D. Manually tag each column containing PII using the Snowflake web UI or the 'ALTER TABLE ... ALTER COLUMN ... SET TAG' command. Train data stewards to identify and tag new columns.

E. Implement a stored procedure that leverages external functions to call a Python script hosted on AWS Lambda, which uses a machine learning model to identify PII and apply Snowflake tags.

Answer: C

Explanation:
Option C is the most effective because it leverages Snowflake's native event capture mechanisms (Event Tables, Streams and Tasks) to react to DDL changes in near real-time. This approach is automated, scalable, and avoids the overhead of periodic polling. Options A and B involve periodic scanning which is less efficient. Option D is manual and doesn't scale. Option E introduces unnecessary complexity with external functions and ML models for a relatively simple task, increasing operational overhead.

NO.6 You are using the Snowflake REST API to insert data into a table named 'RAW JSON DATA. The JSON data is complex and nested, and you want to efficiently parse and flatten it into a relational structure. You have the following JSON sample:

```
{
  "id": "123",
  "event_type": "purchase",
  "user": {
    "user_id": "456",
    "email": "user@example.com",
    "address": {
      "street": "123 Main St",
      "city": "Anytown"
    }
  },
  "items": [
    {"item_id": "789", "price": 10.00},
    {"item_id": "101", "price": 20.00}
  ]
}
```

Which SQL statement, executed after loading the raw JSON using the REST API, is the MOST efficient way to flatten the JSON and extract relevant fields into a new table named 'PURCHASES' with columns like 'EVENT TYPE', 'USER D', 'EMAIL', 'STREET, 'CITY', 'ITEM ID', and 'PRICE'?

```
CREATE TABLE PURCHASES AS SELECT RAW_JSON_DATA:id::$STRING, RAW_JSON_DATA:event_type::$STRING, RAW_JSON_DATA:user.user_id::$STRING, RAW_JSON_DATA:user.email::$STRING, RAW_JSON_DATA:user.address.street::$STRING, RAW_JSON_DATA:user.address.city::$STRING, item.value:item_id::$STRING, item.value:price::$NUMBER FROM RAW_JSON_DATA, LATERAL FLATTEN(input => RAW_JSON_DATA:items) item;
```

```
CREATE TABLE PURCHASES AS SELECT GET_PATH(RAW_JSON_DATA, 'id'), GET_PATH(RAW_JSON_DATA, 'event_type'), GET_PATH(RAW_JSON_DATA, 'user.user_id'), GET_PATH(RAW_JSON_DATA, 'user.email'), GET_PATH(RAW_JSON_DATA, 'user.address.street'), GET_PATH(RAW_JSON_DATA, 'user.address.city') FROM RAW_JSON_DATA;
```

```
CREATE TABLE PURCHASES AS SELECT RAW_JSON_DATA:id::VARCHAR, RAW_JSON_DATA:event_type::VARCHAR, RAW_JSON_DATA:user:user_id::VARCHAR, RAW_JSON_DATA:user:email::VARCHAR, RAW_JSON_DATA:user:address:street::VARCHAR, RAW_JSON_DATA:user:address:city::VARCHAR, item.value:item_id::VARCHAR, item.value:price::NUMBER FROM RAW_JSON_DATA, TABLE(FLATTEN(input => RAW_JSON_DATA:items)) item;
```

```
CREATE TABLE PURCHASES AS SELECT RAW_JSON_DATA[0]:id, RAW_JSON_DATA[0]:event_type, RAW_JSON_DATA[0]:user.user_id, RAW_JSON_DATA[0]:user.email, RAW_JSON_DATA[0]:user.address.street, RAW_JSON_DATA[0]:user.address.city FROM RAW_JSON_DATA;
```

```
CREATE TABLE PURCHASES AS SELECT PARSE_JSON(RAW_JSON_DATA):id::$STRING, PARSE_JSON(RAW_JSON_DATA):event_type::$STRING, PARSE_JSON(RAW_JSON_DATA):user.user_id::$STRING, PARSE_JSON(RAW_JSON_DATA):user.email::$STRING, PARSE_JSON(RAW_JSON_DATA):user.address.street::$STRING, PARSE_JSON(RAW_JSON_DATA):user.address.city::$STRING FROM RAW_JSON_DATA;
```

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E
Answer: C
Explanation:
Option C is the most efficient and correct. It uses the correct Snowflake syntax for accessing JSON elements with and for data type casting and also correctly uses the TABLE(FLATTEN()) function to handle the nested 'items' array. Option A utilizes a deprecated syntax LATERAL FLATTEN'. Options B, D and E don't handle the nested JSON structure properly or the flattening of the 'items' array, resulting in incomplete or incorrect data extraction.

NO.7 A Data Engineer needs to implement dynamic data masking for a PII column named in a table 'CUSTOMERS. The masking policy should apply only to users with the role 'ANALYST. If the user is not an 'ANALYST, the full 'EMAIL' address should be displayed. Which of the following is the MOST efficient and secure way to achieve this using Snowflake's masking policies?

```
CREATE MASKING POLICY email_mask AS (val VARCHAR) RETURNS VARCHAR -> CASE WHEN CURRENT_ROLE() = 'ANALYST' THEN 'XXXXXXXXXX' ELSE val END; ALTER TABLE CUSTOMERS MODIFY COLUMN EMAIL SET MASKING POLICY email_mask;
```

```
CREATE MASKING POLICY email_mask AS (val VARCHAR) RETURNS VARCHAR -> CASE WHEN IS_ROLE_IN_SESSION('ANALYST') THEN 'XXXXXXXXXX' ELSE val END; ALTER TABLE CUSTOMERS MODIFY COLUMN EMAIL SET MASKING POLICY email_mask;
```

```
CREATE MASKING POLICY email_mask AS (val VARCHAR) RETURNS VARCHAR -> CASE WHEN CURRENT_ROLE() LIKE '%ANALYST%' THEN 'XXXXXXXXXX' ELSE val END; ALTER TABLE CUSTOMERS MODIFY COLUMN EMAIL SET MASKING POLICY email_mask;
```

```
CREATE MASKING POLICY email_mask AS (val VARCHAR) RETURNS VARCHAR -> CASE WHEN SYSTEM$GET_PRIVILEGE('ANALYST', CURRENT_USER()) = 'GRANTED' THEN 'XXXXXXXXXX' ELSE val END; ALTER TABLE CUSTOMERS MODIFY COLUMN EMAIL SET MASKING POLICY email_mask;
```

```
CREATE MASKING POLICY email_mask AS (val VARCHAR) RETURNS VARCHAR -> CASE WHEN EXISTS (SELECT 1 FROM TABLE(SHOW GRANTS TO USER(CURRENT_USER())) WHERE privilege = 'USAGE' and role = 'ANALYST') THEN 'XXXXXXXXXX' ELSE val END; ALTER TABLE CUSTOMERS MODIFY COLUMN EMAIL SET MASKING POLICY email_mask;
```

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E
Answer: B
Explanation:
Using is the most efficient and recommended approach for checking a user's active role within a masking policy. It avoids unnecessary parsing or string comparisons. It directly checks if the role is

active in the current session, providing a clear and performant check. The other options either rely on string comparisons, which can be less reliable, or are overly complex and inefficient ways to determine role membership.

NO.8 You have a data pipeline that loads data from an internal stage into a Snowflake table Craw_data'). The pipeline is experiencing intermittent failures with the error 'SQL compilation error: Stage 'MY INTERNAL STAGE' is immutable'. What are the potential causes of this error and how would you troubleshoot it?

A. The internal stage has been accidentally dropped and recreated with the same name during the COPY operation. Verify the stage's existence and creation timestamp.

B. The user executing the COPY INTO command lacks the necessary privileges (USAGE on the stage). Grant the appropriate privileges to the user or role.

C. Another concurrent process is attempting to drop or alter the internal stage while the COPY INTO command is running. Implement proper locking mechanisms to prevent concurrent modifications.

D. The internal stage is being used by multiple COPY INTO commands simultaneously, causing a resource contention issue. Implement queuing or throttling mechanisms to manage concurrent data loading.

E. This error is caused by insufficient warehouse size. Increase the warehouse size to accommodate the COPY INTO operation.

Answer: A,C

Explanation:

The 'Stage is immutable' error typically indicates that the stage's definition has changed during the COPY operation. This can happen if the stage is dropped and recreated (option A) or if another process is altering the stage concurrently (option C). Privilege issues (option B) would usually result in a different error message. Resource contention (option D) is less likely to cause this specific error but could impact performance. Warehouse size (option E) is generally not directly related to this error.

NO.9 A data engineering team is using a Snowflake stream to capture changes made to a source table named 'orders'. They want to only capture 'INSERT and 'UPDATE operations but exclude 'DELETE operations from being captured in the stream. Which of the following configurations will achieve this requirement? Assume the stream has already been created and is named 'orders_stream'.

A. It's impossible to configure a stream to exclude specific DML operations. All changes are always tracked.

B. Create a view on top of the base table that filters out deleted rows, and then create a stream on the view.

C. Alter the stream using the 'HIDE_DELETES parameter: 'ALTER STREAM orders_stream SET HIDE_DELETES = TRUE;'

D. Use task and stream combination. In the task, create view using 'select from orders where metadata$isDelete = false' and create stream on that view.

E. Create a Snowflake task that periodically truncates the stream's metadata table, removing DELETE records.

Answer: C

Explanation:

Snowflake streams can be configured to hide delete operations using the parameten Setting 'HIDE_DELETES = TRUE' will prevent delete operations from being exposed through the stream. Option A is incorrect as streams can be configured. Option B, while functional, adds an extra layer of complexity. Option D doesn't exist as a valid parameter for streams. Option E is a highly unconventional and unsupported approach.

NO.10 You are designing a data pipeline to ingest streaming data from Kafka into Snowflake. The data contains nested JSON structures representing customer orders. You need to transform this data and load it into a flattened Snowflake table named 'ORDERS FLAT'. Given the complexities of real-time data processing and the need for custom logic to handle certain edge cases within the JSON payload, which approach provides the MOST efficient and maintainable solution for transforming and loading this streaming data into Snowflake?

A. Use Snowflake's Snowpipe with a COPY INTO statement that utilizes the 'STRIP OUTER ARRAY option to handle the JSON array, combined with a series of SQL queries with 'LATERAL FLATTEN' functions to extract the nested data after loading into a VARIANT column.

B. Implement a custom external function (UDF) written in Java to parse and transform the JSON data before loading it into Snowflake. Configure Snowpipe to call this UDF during the data ingestion process. This UDF will flatten the JSON structure and return a tabular format directly insertable into 'ORDERS FLAT.

C. Utilize a third-party ETL tool (like Apache Spark) to consume the data from Kafka, perform the JSON flattening and transformation logic, and then use the Snowflake connector to load the data into the 'ORDERS FLAT' table in batch mode.

D. Create a Python UDF that calls 'json.loads()' to parse the JSON within Snowflake and then use SQL commands with 'LATERAL FLATTEN' to navigate and extract the desired fields into a staging table. Afterward, use a separate SQL script to insert from staging to the final table 'ORDERS FLAT

E. Use Snowflake's built-in JSON parsing functions within a Snowpipe COPY INTO statement, combined with a 'CREATE VIEW' statement on top of the loaded data. The view will use 'LATERAL FLATTEN' to present the data in the desired flattened structure without physically transforming the underlying data.

Answer: B

Explanation:

Option B offers the most efficient and maintainable solution. Using a Java UDF allows complex JSON parsing and transformation logic to be encapsulated and optimized. Calling the UDF directly from Snowpipe ensures efficient real-time processing during data ingestion. While other options can achieve the result, they often involve unnecessary steps or performance overhead (e.g., loading into a VARIANT column and then flattening with SQL, using external ETL tools for streaming ingestion, or creating views instead of physically transforming the data).

NO.11 A data engineering team is running a series of complex analytical queries against a large Snowflake table. They notice that query performance is inconsistent, with some queries running much slower than others. After investigation, they determine that the queries are not properly leveraging the data clustering. Which of the following actions could improve the query performance related to the data clustering? Select all that apply.

A. Run 'ALTER TABLE CLUSTER BY (columnl, column2Y to explicitly define a clustering key on the table.

B. Run 'ALTER TABLE DROP CLUSTERING KEY to remove the existing clustering key and allow Snowflake to automatically cluster the data.
C. Run 'SHOW TABLES LIKE ' ' and review the 'clustering_information' column to understand the current clustering depth and benefit.
D. Increase the virtual warehouse size to improve the performance of all queries, regardless of clustering.
E. Execute SYSTEM$CLUSTERING INFORMATION(')' y to assess the impact of clustering on query performance and determine if re-clustering is needed.
Answer: A,C,E
Explanation:
Option A is correct because explicitly defining a clustering key allows Snowflake to optimize data organization for common query patterns. Option C is correct because understanding the current clustering state helps diagnose if clustering is effective. Option E is correct, using SYSTEM$CLUSTERING INFORMATION is a crucial method to determine reclustering needs. Option B can worsen performance, especially if the automatic clustering does not align with common query patterns. Option D may improve performance to a small extend but doesn't directly address the clustering issue.

NO.12 You have a table 'ORDERS in your Snowflake database. You are implementing a new data transformation pipeline. Before deploying the pipeline to production, you want to validate the changes in a development environment. You decide to use Time Travel to create a snapshot of the 'ORDERS' table before the transformation and compare it with the transformed data'. Which sequence of SQL commands would best facilitate this validation, assuming your development database and schema structure mirrors production?
A.
```
-- In Production
CREATE OR REPLACE TABLE ORDERS_BEFORE_TRANSFORMATION AS SELECT   FROM ORDERS BEFORE(STATEMENT => LAST_QUERY_ID());
-- Run Transformation
-- In Development
CREATE OR REPLACE TABLE ORDERS_AFTER_TRANSFORMATION AS SELECT   FROM ORDERS;
-- Compare
SELECT   FROM ORDERS_BEFORE_TRANSFORMATION EXCEPT SELECT   FROM ORDERS_AFTER_TRANSFORMATION;
```
B.
```
-- In Production
CREATE OR REPLACE TABLE ORDERS_BEFORE_TRANSFORMATION AS SELECT   FROM ORDERS BEFORE(TIMESTAMP => CURRENT_TIMESTAMP());
-- Run Transformation
-- In Development
CREATE OR REPLACE TABLE ORDERS_AFTER_TRANSFORMATION AS SELECT   FROM ORDERS;
-- Compare
SELECT   FROM ORDERS_BEFORE_TRANSFORMATION EXCEPT SELECT   FROM ORDERS_AFTER_TRANSFORMATION;
```
C.
```
-- In Production
CREATE OR REPLACE TABLE ORDERS_BEFORE_TRANSFORMATION AS SELECT   FROM ORDERS BEFORE(STATEMENT => LAST_QUERY_ID());
-- Run Transformation
-- Clone to development
CREATE OR REPLACE TABLE DEV_DB.PUBLIC.ORDERS_AFTER_TRANSFORMATION CLONE PROD_DB.PUBLIC.ORDERS;
-- Compare
SELECT   FROM ORDERS_BEFORE_TRANSFORMATION EXCEPT SELECT   FROM DEV_DB.PUBLIC.ORDERS_AFTER_TRANSFORMATION;
```
D.

```
-- In Production
CREATE OR REPLACE TABLE ORDERS_BEFORE_TRANSFORMATION AS SELECT   FROM ORDERS BEFORE(STATEMENT => LAST_QUERY_ID());
-- Run Transformation
-- Clone the original table to development
CREATE OR REPLACE TABLE DEV_DB.PUBLIC.ORDERS_BEFORE_TRANSFORMATION CLONE PROD_DB.PUBLIC.ORDERS BEFORE (STATEMENT => LAST_QUERY_ID());
-- Clone the transformed table to development
CREATE OR REPLACE TABLE DEV_DB.PUBLIC.ORDERS_AFTER_TRANSFORMATION CLONE PROD_DB.PUBLIC.ORDERS;
-- Compare
SELECT   FROM DEV_DB.PUBLIC.ORDERS_BEFORE_TRANSFORMATION EXCEPT SELECT   FROM DEV_DB.PUBLIC.ORDERS_AFTER_TRANSFORMATION;
```

E.

```
-- In Production
CREATE OR REPLACE TABLE ORDERS_BEFORE_TRANSFORMATION AS SELECT   FROM ORDERS BEFORE(TIMESTAMP => dateadd(hour, -1, current_timestamp()));
-- Run Transformation
-- In Development
CREATE OR REPLACE TABLE ORDERS_AFTER_TRANSFORMATION AS SELECT   FROM ORDERS;
-- Compare
SELECT   FROM ORDERS_BEFORE_TRANSFORMATION EXCEPT SELECT   FROM ORDERS_AFTER_TRANSFORMATION;
```

Answer: D

Explanation:

Option D is the most complete and reliable approach. It first creates a backup table in production using Time Travel before the transformation. Then, clones both the original (pre-transformation) table and the transformed table into the development environment. Finally, it compares these cloned tables to validate the transformation. The use of LAST_QUERY_ID() would not be suitable since the clone is required in the same session, while TIMESTAMP based approach is less reliable due to the lack of synchronisation on when the query was executed.

NO.13 A data engineering team is implementing column-level security on a Snowflake table named 'CUSTOMER DATA containing sensitive PII. They want to mask the 'EMAIL' column for users in the 'ANALYST role but allow users in the 'DATA SCIENTIST role to view the unmasked email addresses. The 'ANALYST role already has SELECT privileges on the table. Which of the following steps are necessary to achieve this using a masking policy?

A. Create a masking policy that uses the CURRENT ROLE() function to return a masked value if the current role is 'ANALYST and the original value otherwise.

B. Create a masking policy that uses the IS_ROLE_IN_SESSION('ANALYST') function to return a masked value if the analyst role is active in current session and the original value otherwise.

C. Create a masking policy that uses the CURRENT_USER() function to check if the current user belongs to the 'ANALYST' role.

D. Create a masking policy with a CASE statement that checks the CURRENT ROLE() function to see if it's 'ANALYST'. If true, mask the email; otherwise, return the original email.

E. Create a dedicated view on 'CUSTOMER DATA' for analysts with the 'EMAIL' column masked using a CASE statement within the view's SELECT statement. Grant SELECT privilege to the ANALYST role on the view only.

Answer: A,D

Explanation:

Options A and D are correct. They both use the 'CURRENT ROLE()' function within the masking policy to conditionally mask the email addresses based on the active role. Option B uses which checks if a role has been granted to the user which is not the intention. CURRENT_USER is not appropriate because the requirement is role-based. Creating a view (option E) is a valid but less scalable and maintainable solution compared to a masking policy. Creating a view will require more maintenance than creating a policy which is more integrated and scalable.

NO.14 You're designing a data masking solution for a 'CUSTOMER' table with columns like 'CUSTOMER ID', 'NAME', 'EMAIL', and 'PHONE NUMBER. You want to implement the following requirements: 1. The 'SUPPORT' role should be able to see the last four digits of the 'PHONE NUMBER and a hashed version of the 'EMAIL'. 2. The 'MARKETING' role should be able to see the full 'NAME' and a domain-only version of the 'EMAIL' (everything after the '@' symbol). 3. All other roles should see masked values for 'EMAIL' and 'PHONE NUMBER. Which of the following masking policy definitions BEST achieves these requirements using Snowflake's built-in functions and RBAC?

A.

```
CREATE OR REPLACE MASKING POLICY customer_email_mask AS (email VARCHAR) RETURNS VARCHAR ->
  CASE
    WHEN CURRENT_ROLE() = 'SUPPORT' THEN SHA2(email)
    WHEN CURRENT_ROLE() = 'MARKETING' THEN SUBSTRING(email, INSTR(email, '@') + 1)
    ELSE 'MASKED'
  END;

CREATE OR REPLACE MASKING POLICY customer_phone_mask AS (phone VARCHAR) RETURNS VARCHAR ->
  CASE
    WHEN CURRENT_ROLE() = 'SUPPORT' THEN 'XXXXXXXXXX' || RIGHT(phone, 4)
    ELSE 'MASKED'
  END;
```

B.

```
CREATE OR REPLACE MASKING POLICY customer_email_mask AS (email VARCHAR) RETURNS VARCHAR ->
    CASE
        WHEN CURRENT_ROLE() = 'SUPPORT' THEN HASH(email)
        WHEN CURRENT_ROLE() = 'MARKETING' THEN SPLIT_PART(email, '@', 2)
        ELSE 'MASKED'
    END;

CREATE OR REPLACE MASKING POLICY customer_phone_mask AS (phone VARCHAR) RETURNS VARCHAR ->
    CASE
        WHEN CURRENT_ROLE() = 'SUPPORT' THEN CONCAT('XXXXXXXXXX', RIGHT(phone, 4))
        ELSE 'MASKED'
    END;
```

C.

```
CREATE OR REPLACE MASKING POLICY customer_email_mask AS (email VARCHAR) RETURNS VARCHAR ->
  CASE
    WHEN CURRENT_ROLE() = 'SUPPORT' THEN SHA2(email)
    WHEN CURRENT_ROLE() = 'MARKETING' THEN SPLIT_PART(email, '@', 2)
    ELSE 'MASKED'
  END;

CREATE OR REPLACE MASKING POLICY customer_phone_mask AS (phone VARCHAR) RETURNS VARCHAR ->
  CASE
    WHEN CURRENT_ROLE() = 'SUPPORT' THEN RPAD('XXXXXXXXXX',LENGTH(phone)-4,'X') || RIGHT(phone, 4)
    ELSE 'MASKED'
  END;
```

D.

```
CREATE OR REPLACE MASKING POLICY customer_email_mask AS (email VARCHAR) RETURNS VARCHAR ->
    CASE
        WHEN CURRENT_ROLE() = 'SUPPORT' THEN SHA2(email)
        WHEN CURRENT_ROLE() = 'MARKETING' THEN SUBSTRING(email, POSITION('@' IN email) + 1)
        ELSE 'MASKED'
    END;

CREATE OR REPLACE MASKING POLICY customer_phone_mask AS (phone VARCHAR) RETURNS VARCHAR ->
    CASE
        WHEN CURRENT_ROLE() = 'SUPPORT' THEN CONCAT(REPEAT('X', LENGTH(phone)-4), RIGHT(phone, 4))
        ELSE 'MASKED'
    END;
```

E.

```
CREATE OR REPLACE MASKING POLICY customer_email_mask AS (email VARCHAR) RETURNS VARCHAR ->
  CASE
    WHEN CURRENT_ROLE() = 'SUPPORT' THEN HASH(email)
    WHEN CURRENT_ROLE() = 'MARKETING' THEN SUBSTRING(email, POSITION('@' IN email) + 1)
    ELSE 'MASKED'
  END;

CREATE OR REPLACE MASKING POLICY customer_phone_mask AS (phone VARCHAR) RETURNS VARCHAR ->
  CASE
    WHEN CURRENT_ROLE() = 'SUPPORT' THEN CONCAT(REPEAT('X', LENGTH(phone)-4), RIGHT(phone, 4))
    ELSE 'MASKED'
  END;
```

Answer: D

Explanation:

Option D is the best solution as it effectively uses 'SHA2 for the hashed email for support, 'SUBSTRING(email, POSITION('@' IN email) + 1)' correctly extracts the domain, and LENGTH(phone)-4), RIGHT(phone, 4))' creates the 'XXXXXXXXXX' and then the final four digits of the phone. Other options are not correct because they may use incorrect functions, or because they use outdated syntax (CONCAT instead of 'IF). The correct solution uses the correct functions, SHA2()' for Email Hash for support User, 'SUBSTRING(email, POSITION('@' IN email) + 1)' extract Domain name of the Email for Marketing User, LENGTH(phone)-4), RIGHT(phone, 4))' masking the Phone number by preserving last four digits for Support User.

NO.15 A data engineering team has created a Snowflake Listing to share their company's sales data'. They want to allow consumers to access the Listing programmatically. The consumers need to know when new versions of the Listing are available. What is the MOST efficient method for consumers to be notified about new Listing versions without continuously polling Snowflake?

A. The data provider manually sends email notifications to each consumer whenever a new version is created.

B. The consumer sets up a scheduled task in Snowflake to periodically query the 'SHOW VERSIONS command for the Listing.

C. Snowflake automatically notifies consumers via an event notification service (e.g., AWS SNS, Azure Event Grid) integrated with the Data Marketplace when a new version is available.

D. The data consumer implements a custom webhook endpoint that the data provider calls

whenever a new Listing version is published. The data provider will need to maintain a list of all consumers.

Answer: C

Explanation:

Snowflake's integration with event notification services like AWS SNS or Azure Event Grid provides the most efficient and scalable way to notify consumers about new Listing versions. This eliminates the need for manual intervention or continuous polling, making it a push-based system. Snowflake automatically handles the notification process based on preconfigured event triggers.

NO.16 You're building a data product on the Snowflake Marketplace that includes a view that aggregates data from a table containing Personally Identifiable Information (PII). You need to ensure that consumers of your data product CANNOT directly access the underlying PII data but can only see the aggregated results from the view. What is the MOST secure and recommended approach to achieve this?

A. Grant the 'SELECT privilege directly on the underlying PII table to the share used for the Marketplace listing, along with the 'SELECT privilege on 'sensitive data view'.

B. Grant the 'SELECT privilege only on the to the share used for the Marketplace listing. Do not grant any privileges on the underlying PII table.

C. Create a stored procedure that returns the aggregated data, and grant EXECUTE privilege on the stored procedure to the share. The stored procedure SELECTs from the PII table.

D. Grant USAGE privilege on the database containing the PII table and to the share.

E. Grant 'READ privilege on the internal stage containing the data files backing the PII table.

Answer: B

Explanation:

Granting only 'SELECT privilege on the (option B) ensures that consumers can only access the view and not the underlying PII data. Granting 'SELECT on the underlying table (option A) defeats the purpose of the view. Using a stored procedure (option C), while potentially masking the data access, is less performant and can still expose data if not carefully implemented. 'USAGE privilege (option D) only allows access to the database, not the data itself. 'READ' on the stage (option E) allows direct access to the raw data, which exposes the PII.

NO.17 A financial institution needs to tokenize sensitive customer data (credit card numbers) stored in a Snowflake table named 'CUSTOMER_DATA before it's consumed by a downstream reporting application. The institution uses an external tokenization service accessible via a REST API. Which of the following approaches is the MOST secure and scalable way to implement tokenization during data loading, minimizing exposure of the raw credit card data within Snowflake?

A. Use a Snowflake UDF (User-Defined Function) written in Java that calls the external tokenization API directly. Create a masking policy that utilizes the UDF and applies it to the credit card number column.

B. Load the raw data into a staging table, then create a Snowflake Task that executes a stored procedure. The stored procedure calls the external tokenization API using 'SYSTEM$EXTERNAL_FUNCTION_REQUEST' for each row and updates the original table with the tokenized values.

C. Load the raw data directly into the 'CUSTOMER DATA' table. Create a masking policy that utilizes a UDF that calls the external tokenization API directly to tokenize the credit card number values on

read.

D. Use Snowflake's Data Sharing feature to securely share the raw data with the downstream application, instructing them to perform the tokenization within their own environment.

E. Utilize Snowflake's Snowpipe to ingest the data directly. Inside a COPY INTO statement, use an external function to call the tokenization service during the ingestion process to tokenize the data before it's loaded into the target table.

Answer: E

Explanation:

Option E is the most secure and scalable approach. It tokenizes the data during the load process, minimizing the amount of time the raw data resides in Snowflake. Using a UDF in a masking policy (options A and C) tokenizes the data on read, meaning the raw data is stored in Snowflake. Option B, using a stored procedure and , can be less efficient for large datasets. Data sharing raw data (Option D) defeats the purpose of tokenization for the source environment.

NO.18 You have a 'SALES table and a 'PRODUCTS table. The 'SALES table contains daily sales transactions, including 'SALE DATE , 'PRODUCT ID', and 'QUANTITY. The 'PRODUCTS table contains 'PRODUCT and 'CATEGORY. You need to create a materialized view to track the total quantity sold per category daily, optimized for fast query performance. You anticipate frequent updates to the 'SALES table but infrequent changes to the 'PRODUCTS table. Which of the following strategies would provide the MOST efficient materialized view implementation, considering both data freshness and query performance?

A. Create a standard materialized view that joins 'SALES' and 'PRODUCTS' , grouping by 'SALE DATE and 'CATEGORY without any specific clustering key.

B. Create a standard materialized view that joins 'SALES' and 'PRODUCTS' , grouping by 'SALE_DATE and 'CATEGORY, and defining a clustering key on 'SALE DATE.

C. Create a standard materialized view that joins 'SALES' and 'PRODUCTS' , grouping by 'SALE_DATE and 'CATEGORY, and defining a clustering key on ' CATEGORY.

D. Create a standard materialized view that joins 'SALES' and 'PRODUCTS' , grouping by 'SALE_DATE and 'CATEGORY, and defining a clustering key on 'SALE DATE' and 'CATEGORY.

E. Create two materialized views: one for daily sales by product and another joining the first with 'PRODUCTS' to aggregate by category. Cluster the first view by 'SALE DATE' and the second by 'CATEGORY'.

Answer: B

Explanation:

Option B is most efficient. Clustering the materialized view on 'SALE_DATE will significantly improve query performance when filtering or grouping by date, which is a common operation in time-series data. Although frequent updates will affect the maintenance costs of the materialized view, querying on date will be very efficient. Option A is less efficient due to the lack of clustering. Option C may not be the best choice if filtering/grouping primarily occurs on date. Option D is also good, but Option B is better if most of the query filter is on SALE DATE. Option E introduces complexity and two refreshes may create a delay in data available.

NO.19 You are developing a Snowpark Python application that needs to process data from a Kafka topic. The data is structured as Avro records. You want to leverage Snowpipe for ingestion and Snowpark DataFrames for transformation. What is the MOST efficient and scalable approach to

integrate these components?

A. Create a Kafka connector that directly writes Avro data to a Snowflake table. Then, use Snowpark DataFrames to read and transform the data from that table.

B. Use Snowpipe to ingest the Avro data to a raw table stored as binary. Then, use a Snowpark Python UDF with an Avro deserialization library to convert the binary data to a Snowpark DataFrame.

C. Configure Snowpipe to ingest the raw Avro data into a VARIANT column in a staging table. Utilize a Snowpark DataFrame with Snowflake's get_object field function on the variant to get an object by name, and create columns based on each field.

D. Convert Avro data to JSON using a Kafka Streams application before ingestion. Use Snowpipe to ingest the JSON data to a VARIANT column and then process it using Snowpark DataFrames.

E. Create external functions to pull the Avro data into a Snowflake stage and then read the data with Snowpark DataFrames for transformation.

Answer: D

Explanation:

Option D is generally the most efficient. Converting Avro to JSON before ingestion simplifies the integration with Snowpipe and Snowpark. Snowpipe is optimized for semi-structured data like JSON within a VARIANT column. Subsequently, Snowpark DataFrames can easily process the JSON data using built-in functions, avoiding the complexity and potential performance bottlenecks of UDFs (Option B) or custom connectors (Option A). Although Snowflake's function can work with variant data, operating on raw Avro data is not natively supported by Snowpipe without pre-processing or complex UDF logic. External functions (Option E) add another layer of complexity for data retrieval.

NO.20 You are designing a data pipeline using Snowpipe to ingest data from multiple S3 buckets into a single Snowflake table. Each S3 bucket represents a different data source and contains files in JSON format. You want to use Snowpipe's auto-ingest feature and a single Snowpipe object for all buckets to simplify management and reduce overhead. However, each data source has a different JSON schem a. How can you best achieve this goal while ensuring data is loaded correctly and efficiently into the target table?

A. Create a separate Snowpipe for each S3 bucket. Although this creates more Snowpipe objects, it allows you to specify a different FILE FORMAT and transformation logic for each data source.

B. Use a single Snowpipe with a generic FILE FORMAT that can handle all possible JSON schemas. Implement a VIEW on top of the target table to transform and restructure the data based on the source bucket.

C. Use a single Snowpipe and leverage Snowflake's VARIANT data type to store the raw JSON data. Create separate external tables, each pointing to a specific S3 bucket, and use SQL queries to transform and load the data into the target table.

D. Use a single Snowpipe and leverage Snowflake's ability to call a user-defined function (UDF) within the 'COPY INTO' statement to transform the data based on the S3 bucket path. The UDF can parse the bucket path and apply the appropriate JSON schema transformation.

E. Since Snowpipe cannot handle multiple schemas with a single pipe, pre-process the data in S3 using an AWS Lambda function to transform all files into a common schema before they are ingested by the Snowpipe.

Answer: D

Explanation:

The most efficient and manageable approach is to use a single Snowpipe with a UDF to handle schema variations. The UDF can inspect the S3 bucket path (available as metadata within the 'COPY INTO' statement) and apply the correct transformation logic for each data source. Creating separate Snowpipes (A) adds unnecessary overhead. Using a generic 'FILE FORMAT and a VIEW (B) might work for simple transformations, but it becomes complex with significant schema differences. Using VARIANT and external tables (C) defeats the purpose of Snowpipe. Pre-processing in S3 (E) adds complexity outside of Snowflake. UDF provides schema flexibility during ingest and leverages Snowpipe's capabilities directly.

NO.21 A daily process loads data into a Snowflake table named 'TRANSACTIONS using a COPY INTO statement. The table is clustered on 'TRANSACTION DATE'. Over time, you observe a significant degradation in query performance when querying data within specific date ranges. Analyzing the 'SYSTEM$CLUSTERING INFORMATION' function output for the 'TRANSACTIONS' table reveals a low 'effective clustering_ratio' and a high 'average_overlaps'. Which combination of actions below would BEST address the performance degradation and improve query efficiency?

A. Recluster the table using 'ALTER TABLE TRANSACTIONS RECLUSTER$ and adjust the virtual warehouse size to maximize resource allocation during the recluster operation.

B. Drop the existing clustering key on 'TRANSACTION_DATE, then recreate it with a different clustering key such as 'HASH(TRANSACTION_ID)'.

C. Create a new table with the desired clustering and load data using 'CREATE TABLE AS SELECT statement.

D. Implement a data maintenance schedule that regularly reclusters the table using 'ALTER TABLE TRANSACTIONS RECLUSTER;' during off-peak hours and monitor the 'SYSTEM$CLUSTERING INFORMATION' function periodically.

E. Drop the current clustered table and create a new table with partition by clauses

Answer: A,D

Explanation:

A low 'effective_clustering_ratio' and high 'average_overlaps' indicate that the data is not well-clustered, leading to inefficient query performance. Reclustering the table (A) reorganizes the data based on the clustering key, improving clustering. Creating a schedule (D) ensures that the table remains well-clustered over time. Dropping the clustering key and recreating it with a hash on (B) is unlikely to improve performance for date-range queries. Creating new table via 'CREATE TABLE AS SELECT statement is not the right way. In addition, 'Partition By' clause doesn't exist in Snowflake.

NO.22 You are tasked with ingesting a large volume of CSV files from an external stage into a Snowflake table. Some of these CSV files contain corrupted records with inconsistent delimiters or missing values. You need to ensure that only valid records are loaded into the table, and the corrupted records are captured for further analysis. Which of the following COPY INTO options would BEST address this requirement?

A. Option A

B. Option B

C. Option C

D. Option D

E. Option E

Answer: E

Explanation:
Option E is the most comprehensive solution. ERROR = CONTINUE allows the COPY INTO statement to proceed despite errors. 'SKIP HEADER = 1' handles potential header issues in corrupted files. MISMATCH = FALSE' allows for varying column counts. 'VALIDATION _ MODE' and 'RESULT provide a mechanism to capture and analyze the rejected records, satisfying the requirement to analyze corrupted records. Options A, B, C, and D are insufficient for capturing corrupted records for analysis or may lead to data loss.

NO.23 You have a requirement to create a UDF in Snowflake that transforms data based on a complex set of rules defined in an external Python library. The library requires specific dependencies. You also need to ensure the UDF is secure and that the code is not visible to unauthorized users. Which of the following steps MUST be taken to achieve this?

A. Create a Snowflake Anaconda environment specifying the required Python library dependencies. Then, create a Python UDF, reference the Anaconda environment, and use the 'SECURE' keyword.

B. Upload the Python library and its dependencies as internal stages. Create a Java UDF that executes the Python code using the 'ProcessBuilder' class. Mark the Java UDF as 'SECURE'

C. Create a Python UDF and directly upload the Python library code into the UDF's body. Snowflake automatically manages dependencies for UDFs.

D. Create an external function pointing to an AWS Lambda function or Azure Function that hosts the Python code and its dependencies. Secure the external function using API integration and role-based access control.

E. Package all the Python libaries code into one file, then create an Javascript UDF and load/execute the python code inside the Javascript UDF.

Answer: A
Explanation:
Using Snowflake Anaconda environments allows you to manage Python dependencies for UDFs. Creating a Python UDF referencing the environment and using the 'SECURE keyword ensures both dependency management and code protection. Uploading libraries as internal stages and using Java UDFs is an unnecessarily complex approach. Snowflake does not automatically manage dependencies; they must be explicitly specified through Anaconda. Creating a Python inside a Javascript UDF is not a supported pattern

NO.24 Given the following scenario: You have an external table 'EXT SALES in Snowflake pointing to a data lake in Azure Blob Storage. The storage account network rules are configured to only allow specific IP addresses and virtual network subnets, enhancing security. You are getting intermittent errors when querying 'EXT SALES. Which of the following could be the cause(s) and the corresponding solution(s)? Select all that apply.

A. The Snowflake IP addresses used to access the Azure Blob Storage are not whitelisted in the storage account's firewall settings. Solution: Obtain the Snowflake IP address ranges for your region and add them to the storage account's allowed IP addresses.

B. The table function cache is stale, causing access to non-existent files. Solution: Run 'ALTER EXTERNAL TABLE EXT_SALES REFRESH'.

C. The Snowflake service principal does not have the correct permissions on the Azure Blob Storage account. Solution: Ensure the Snowflake service principal has the 'Storage Blob Data Reader' role assigned to it.

D. The file format specified in the external table definition does not match the actual format of the files in Azure Blob Storage. Solution: Update the 'FILE_FORMAT parameter in the external table definition to match the correct file format.

E. The network connectivity between Snowflake and Azure Blob Storage is unstable. Solution: Implement retry logic in your queries to handle transient network errors.

Answer: A,C

Explanation:

Options A and C are the most likely causes. Network restrictions often lead to connectivity issues if Snowflake's IP addresses are not whitelisted (A). Incorrect permissions for the Snowflake service principal (C) will also prevent access to the data lake. Option B is relevant if there are issues around schema changes or new files added. Option D will cause errors all the time not intermittently. Option E Snowflake automatically retries some queries. In a secure environment where IP whitelisting is mandated and IAM roles are properly configured, intermittent failures suggest a permission or a networking issue.

NO.25 You are building a data pipeline in Snowflake that uses an external function to perform sentiment analysis on customer reviews stored in a table named 'CUSTOMER REVIEWS'. The external function 'sentiment_analyzer' is hosted on AWS Lambda and requires an API key for authentication. You want to ensure that the API key is securely passed to the Lambda function and prevent unauthorized access. Which of the following approaches represents the MOST secure and recommended method to manage the API key?

A. Store the API key directly in the external function definition as a string literal within the 'AS' clause.

B. Pass the API key as a parameter to the external function each time it is called.

C. Create a Snowflake secret object to store the API key and reference it in the external function definition using the 'USING' clause and 'SYSTEM$GET SECRET function.

D. Store the API key in a Snowflake table with restricted access and retrieve it within the external function's logic.

E. Embed the API key directly into the AWS Lambda function's environment variables, avoiding any transmission from Snowflake.

Answer: C

Explanation:

Storing the API key directly in the function definition (A) or passing it as a parameter (B) exposes the key. Storing it in a table (D) is also less secure than using Snowflake secrets. While embedding the API key into the AWS Lambda function's environment variables (E) improves security, it doesn't address securing the key during transmission from Snowflake and offers no auditability. The most secure approach is to use a Snowflake secret object (C) to store the API key securely and reference it in the external function definition using the clause and 'SYSTEM$GET SECRET function. This method provides encryption at rest and in transit and allows for centralized management and auditing of secrets.

NO.26 You are developing a JavaScript UDF in Snowflake to perform complex data validation on incoming data'. The UDF needs to validate multiple fields against different criteria, including checking for null values, data type validation, and range checks. Furthermore, you need to return a JSON object containing the validation results for each field, indicating whether each field is valid or not and

providing an error message if invalid. Which approach is the MOST efficient and maintainable way to structure your JavaScript UDF to achieve this?

A. Use a single, monolithic JavaScript function with nested if-else statements to handle all validation logic. Return a JSON string containing the validation results.

B. Create separate JavaScript functions for each validation check (e.g., 'isNull', 'isValidType', 'isWithinRange'). Call these functions from the main UDF and aggregate the results into a JSON object.

C. Utilize a JavaScript library like Lodash or Underscore.js within the UDF to perform data manipulation and validation. Return a JSON string containing the validation results.

D. Define a JavaScript object containing validation rules and corresponding validation functions. Iterate through the object and apply the rules to the input data, collecting the validation results in a JSON object. This object is returned as a string.

E. Directly embed SQL queries within the JavaScript UDF to perform data validation checks using Snowflake's built-in functions. Return a JSON string containing the validation results.

Answer: D

Explanation:

Option D provides the most maintainable and efficient approach. By defining validation rules and corresponding functions in a JavaScript object, you can easily add, modify, or remove validation rules without affecting the core logic of the UDF. This approach promotes code reusability and makes the UDF easier to understand and maintain. Options A leads to unmaintainable code. Options B can be better than A but is still less elegant than D. Option C, while potentially useful for certain tasks, adds unnecessary overhead. Option E is generally discouraged due to performance limitations and the complexity of embedding SQL within JavaScript.

NO.27 You are planning to monetize a dataset on the Snowflake Marketplace. You want to provide potential customers with sample data to evaluate before they purchase a full subscription. Which of the following strategies are valid and recommended for offering a free sample of your data within the Snowflake Marketplace? (Select all that apply)

A. Create a separate share containing a subset (e.g., a smaller number of rows or columns) of the full dataset and offer this share as a free trial listing on the Marketplace.

B. Offer a 'free trial' subscription on the primary listing that automatically expires after a set period (e.g., 7 days), allowing customers to access the full dataset during the trial period. You will need to write custom code to manage trial expiration and data access restrictions based on the trial status.

C. Create a view that filters the dataset based on a sampling algorithm (e.g., 'SAMPLE ROW' clause) and share the view through the Marketplace.

D. Provide the consumer with the script to create a database link to your data, allowing them read-only access to a pre-defined sample table, and then revoke the access after a set period.

E. Upload a sample CSV file to a publicly accessible S3 bucket and provide the link in the Marketplace listing description. Consumers can download and load this data into their own Snowflake account for evaluation.

Answer: A,C

Explanation:

Option A is valid: Creating a separate share with a subset of the data is a common and secure way to offer a free sample. Option C is valid: sharing a view filtered by a sampling algorithm offers a

representative sample without requiring manual data management. Option B is less ideal because managing trial expiration and access controls requires custom coding and increases complexity. Offering consumers a database link (Option D) is a valid concept but less secure than using Snowflake's native sharing mechanisms for trial access. Option E is less integrated with the Marketplace experience; the consumer must leave Snowflake to access the sample data, and there is no tracking of who is accessing the sample.

NO.28 You are designing a data sharing solution where the consumer account needs real-time access to a secure view that aggregates data from several tables in your provider account. The consumer should not be able to see the underlying tables. Which of the following approaches offers the MOST secure and efficient way to implement this data sharing while minimizing the risk of data leakage and performance impact on your provider account?

A. Create a shared database and grant SELECT privilege on the underlying tables directly to the consumer's role.

B. Create a secure view that joins the tables and share only the secure view using a data share.

C. Create a materialized view on top of the tables, refresh it periodically, and share the materialized view.

D. Create a standard view that joins the tables and share the view using a data share. Implement row-level security policies on the underlying tables.

E. Create a UDF that encapsulates the data aggregation logic and share the UDF's result using a data share, calling the UDF on demand.

Answer: B

Explanation:

Secure views are specifically designed for data sharing while protecting the underlying data sources. Sharing the secure view ensures that the consumer only sees the aggregated data and cannot access the underlying tables directly. Options A and D expose the underlying tables, increasing the risk of data leakage. Option C introduces latency due to the materialized view refresh. Option E adds unnecessary complexity and potential performance overhead.

NO.29 You're designing a near real-time data pipeline for clickstream data using Snowpipe Streaming. The data volume is extremely high, with bursts exceeding 1 million events per second. Your team reports intermittent ingestion failures and latency spikes. Considering the constraints of Snowpipe Streaming, which of the following strategies would be MOST effective in mitigating these issues, assuming the data format is optimized and network latency is minimal?

A. Increase the number of Snowflake virtual warehouses to handle the increased load.

B. Implement client-side retry logic with exponential backoff and jitter to handle transient errors and avoid overwhelming the service.

C. Implement a message queue (e.g., Kafka) in front of Snowpipe Streaming to buffer incoming events and smooth out the traffic spikes.

D. Reduce the size of each micro-batch being sent to Snowpipe Streaming to minimize the impact of individual failures.

E. Switch from Snowpipe Streaming to Classic Snowpipe, as it is more resilient to high data volumes.

Answer: B,C

Explanation:

B and C are correct. Implementing client-side retry logic with exponential backoff (B) prevents

overwhelming the service during transient errors. Using a message queue like Kafka (C) buffers the data, smoothing out traffic spikes and providing better resilience. A is less effective as scaling warehouses won't directly address client-side issues like retry logic and buffering. D can help but is not as effective as a buffering mechanism or robust retry strategy. E is incorrect as Snowpipe Streaming is designed for lower latency than classic Snowpipe.

NO.30 You are tasked with creating an external function in Snowflake that calls a REST API. The API requires a bearer token for authentication, and the function needs to handle potential network errors and API rate limiting. Which of the following code snippets demonstrates the BEST practices for defining and securing this external function, including error handling?

○ `CREATE OR REPLACE EXTERNAL FUNCTION api_caller(input VARCHAR) RETURNS VARCHAR RETURNS NULL ON NULL INPUT VOLATILE MAX_BATCH_ROWS = 100 ENDPOINT = 'https://api.example.com/data' AS 'python' IMPORTS = ('@my_stage/api_caller.py') WITH (SECURITY_INTEGRATION = my_security_integration);`

○ `CREATE OR REPLACE EXTERNAL FUNCTION api_caller(input VARCHAR) RETURNS VARCHAR RETURNS NULL ON NULL INPUT VOLATILE MAX_BATCH_ROWS = 100 ENDPOINT = 'https://api.example.com/data' CREDENTIAL = (IDENTITY = 'my_identity', PASSWORD = 'my_password') AS 'python' IMPORTS = ('@my_stage/api_caller.py');`

○ `CREATE OR REPLACE EXTERNAL FUNCTION api_caller(input VARCHAR) RETURNS VARIANT RETURNS NULL ON NULL INPUT VOLATILE MAX_BATCH_ROWS = 100 ENDPOINT = 'https://api.example.com/data' AUTHN_POLICY = (TYPE = 'CUSTOM', SECRET = 'api_secret') AS 'python' IMPORTS = ('@my_stage/api_caller.py'); CREATE OR REPLACE SECRET api_secret TYPE = GENERIC_STRING ENCRYPTED_VALUE = '';`

○ `CREATE OR REPLACE EXTERNAL FUNCTION api_caller(input VARCHAR) RETURNS VARIANT RETURNS NULL ON NULL INPUT VOLATILE MAX_BATCH_ROWS = 100 ENDPOINT = 'https://api.example.com/data' AUTHN_POLICY = (TYPE = 'CUSTOM', SECRET = 'api_secret') AS 'python' IMPORTS = ('@my_stage/api_caller.py') USING (SYSTEM$GET_SECRET('my_database.my_schema.api_secret')); CREATE OR REPLACE SECRET api_secret TYPE = GENERIC_STRING ENCRYPTED_VALUE = '';`

○ `CREATE OR REPLACE EXTERNAL FUNCTION api_caller(input VARCHAR) RETURNS VARCHAR RETURNS NULL ON NULL INPUT VOLATILE MAX_BATCH_ROWS = 100 ENDPOINT = 'https://api.example.com/data' AS 'python' IMPORTS = ('@my_stage/api_caller.py') WITH (SECURITY_INTEGRATION = my_security_integration) CONTEXT_HEADERS = (('Authorization', 'Bearer ' || SYSTEM$GET_SECRET('my_database.my_schema.api_secret'))); CREATE OR REPLACE SECRET api_secret TYPE = GENERIC_STRING ENCRYPTED_VALUE = '';`

A. Option A
B. Option B
C. Option C
D. Option D
E. Option E
Answer: E
Explanation:
Option A uses SECURITY_INTEGRATION, which is suitable for cloud provider-managed security but doesn't directly handle the API key. Option B uses CREDENTIAL, which is deprecated. Option C and D use AUTH POLICY and SECRET, but C doesn't use SYSTEM$GET_SECRET within a 'USING' clause or CONTEXT_HEADERS. Option D uses the 'USING' clause but does not use 'CONTEXT HEADERS to pass the token correctly. Option E is the BEST approach because it utilizes 'SECURITY INTEGRATION' along with 'CONTEXT_HEADERS' to pass the Bearer token securely retrieved from the Snowflake secret, ensuring proper authentication. Using CONTEXT HEADERS allows setting the authorization header directly. Also, its importand to create the 'SECRET api_secret' for this code to work correctly and this options uses it.

NO.31 A data engineer observes that a Snowflake query, used for generating a daily sales report, consistently runs slower each day, despite the dataset size remaining relatively stable. The query joins a large sales table (SALES) with a smaller product dimension table (PRODUCT) on PRODUCT ID. You've already confirmed that virtual warehouse sizing is adequate and data clustering is enabled on SALES(SALE DATE). Analyze the following scenarios and identify the MOST likely cause of the performance degradation and potential solution:
A. The statistics on the PRODUCT table are outdated. Run 'ANALYZE TABLE PRODUCT' to refresh them.

B. The join between SALES and PRODUCT is causing excessive data spill to local storage due to an inefficient join order. Consider using a JOIN hint, specifically a BROADCAST hint on the PRODUCT table (small table).
C. The data clustering on SALES(SALE DATE) is ineffective as the query also filters on PRODUCT CATEGORY, which isn't clustered. Re-cluster SALES by both SALE DATE and PRODUCT CATEGORY.
D. The Snowflake query optimizer is selecting a suboptimal execution plan because the query is complex. Rewrite the query using temporary tables to break down the logic into smaller steps.
E. The virtual warehouse is being overwhelmed by concurrent queries. Implement workload management rules to prioritize the sales report query.
Answer: B
Explanation:
The most likely cause is inefficient join processing leading to data spilling. Broadcasting the smaller PRODUCT table to all nodes improves join performance by avoiding data redistribution. Option A might help, but broadcasting is more impactful. Option C is incorrect because only one column can be explicitly used for clustering. Option D is a last resort; the optimizer should handle a simple join. Option E is less likely if the overall data volume hasn't changed.

NO.32 Your company is implementing data governance policies in Snowflake and wants to automatically classify data to track Personally Identifiable Information (PII). You have defined a classification policy with a tag 'PII' and associated tag values like 'Email', 'CreditCard', and 'SSN'. You want to monitor the usage of PII data'. Which of the following approaches is the MOST efficient way to track access and modifications to columns tagged with the 'PII' tag and any of its tag values?
A. Create a scheduled task to query the INFORMATION SCHEMCOLUMNS view and filter based on the TAG DATABASE, TAG SCHEMA, and TAG NAME columns to identify PII columns, then query the QUERY_HISTORY view and filter based on the identified column names.
B. Create a masking policy that redacts all PII data. Monitor access attempts via the ACCESS HISTORY view. Although it hinders data usability, this approach guarantees no PII data leakage and allows easy monitoring.
C. Enable Snowflake's data governance features, including object tagging and data classification. Then, leverage the ACCESS HISTORY view, filtering based on POLICY _ TAGS. This will show all accesses to data tagged as PII.
D. Implement a custom UDF that intercepts all queries. Inside the UDF, check if any of the accessed tables or columns are tagged with 'PII', and log those queries into a separate logging table for monitoring.
E. Use Snowflake's native data governance capabilities in conjunction with Snowflake Horizon to actively monitor data quality and access patterns, specifically focusing on PII-tagged columns.
Answer: C
Explanation:
The ACCESS HISTORY view, when combined with Snowflake's object tagging and data classification features, provides the MOST efficient way to monitor access to data tagged with specific tags like 'PII'. Filtering based on POLICY _ TAGS allows you to quickly identify accesses related to PII data. Option A is less efficient due to the need for continuous polling and parsing potentially large query histories. Option B excessively restricts data usability. Option D incurs significant overhead and maintenance costs. Option E, while involving Horizon, doesn't primarily rely on Horizon for access

monitoring of tagged data; Horizon is more aligned with data discovery and cataloging.

NO.33 A data engineering team is building a real-time dashboard in Snowflake to monitor website traffic. The dashboard relies on a complex query that joins several large tables. The query execution time is consistently exceeding the acceptable threshold, impacting dashboard responsiveness. Historical data is stored in a separate table and rarely changes. You suspect caching is not being utilized effectively. Which of the following actions would BEST improve the performance of this dashboard and leverage Snowflake's caching features?

A. Use 'RESULT_SCAN' to cache the query result in the user session for subsequent queries. This is especially effective for large datasets that don't change frequently.

B. Materialize the historical data into a separate table that utilizes clustering and indexing for faster query performance. Refresh this table periodically.

C. Create a materialized view that pre-computes the results of the complex query. Snowflake will automatically refresh the materialized view when the underlying data changes.

D. Increase the size of the virtual warehouse. A larger warehouse will have more resources to execute the query, and the results will be cached for a longer period.

E. Replace the complex query with a series of simpler queries. This will reduce the amount of data that needs to be processed at any one time.

Answer: C

Explanation:

Materialized views are the best option in this scenario. They pre-compute the results of the complex query and store them in a separate table. Snowflake automatically refreshes the materialized view when the underlying data changes, ensuring that the dashboard always displays the most up-to-date information. While increasing the virtual warehouse size (D) can help initially, it's a more expensive and less targeted solution. 'RESULT_SCAN' (A) is session-specific and not suitable for persistent caching for a dashboard accessed by multiple users. Materializing the historical data (B) might help, but it doesn't address the core issue of the complex query. Breaking the query into smaller parts (E) might not be efficient and can introduce complexity.

NO.34 You are working with a directory table named associated with an external stage containing a large number of small JSON files. You need to process only the files containing specific sensor readings based on a substring match within their filenames (e.g., files containing 'temperature' in the filename). You also want to load these files into a Snowflake table 'sensor_readings. Consider performance and cost-effectiveness. Which of the following approaches is the MOST efficient and cost-effective to achieve this? Choose TWO options.

A. Use a Python UDF to iterate through the files listed in , filter based on filename, and then load each matching file individually using the Snowflake Python Connector.

B. Create a view on top of the directory table that filters the 'relative_patW based on the substring match, and then use 'COPY INTO' with the 'FILES' parameter to load the filtered files.

C. Use 'COPY INTO' with the 'PATTERN' parameter, constructing a regular expression that includes the substring match against the filename obtained from the directory table's 'relative_path' column.

D. Load all files from the stage using 'COPY INTO' into a staging table, and then use a Snowflake task to filter and move the relevant records into the 'sensor_readingS table.

E. Create a masking policy based on filenames to control which files users can see.

Answer: B,C

Explanation:

Options B and C are the most efficient and cost-effective. Option B (Create a view and use COPY INTO with FILES): Creating a view that filters the directory table allows you to isolate the relevant filenames. Then, using 'COPY INTO' with the 'FILES' parameter pointing to this filtered view directly instructs Snowflake to load only the specified files, minimizing unnecessary data processing. This is efficient as it leverages Snowflake's built-in capabilities. Option C (COPY INTO with the PATTERN parameter): The 'PATTERN' parameter within the 'COPY INTO' command allows you to specify a regular expression. By incorporating the substring match into this regular expression against the metadata$filename" , you can directly filter which files are loaded during the 'COPY INTO operation. This avoids loading irrelevant data and is generally more performant than iterating through files using a UDF. Other options are less efficient or less cost-effective: Option A (Python UDF): Using a Python UDF for this task is generally less efficient. Snowflake is designed to handle this processing natively, and using UDF can lead to performance overhead due to data serialization and deserialization between Snowflake and the UDF environment. Option D (Load all and filter later): Loading all files into a staging table and then filtering is wasteful. It increases data processing time and costs since you're loading unnecessary data. It's always better to filter data closer to the source if possible. Option E (Masking Policy): Masking policies are for security, not data transformation. They are applied at the query level to prevent users from seeing data, but do not help in efficiently processing only specific files.

NO.35 You are a data engineer responsible for data governance in a Snowflake environment. Your company has implemented data classification using tags to identify sensitive data'. The compliance team has requested a report detailing all tables and columns that contain PII data, specifically including the tag name, tag value, the fully qualified name of the table, and the column name. You have the necessary privileges to access the Snowflake metadata views. Which of the following queries would provide the MOST comprehensive and accurate report, considering performance and ease of understanding?

A.

```
SELECT c.table_catalog, c.table_schema, c.table_name, c.column_name, t.tag_name, t.tag_value FROM snowflake.account_usage.columns c JOIN snowflake.account usage.tag references t ON c.table catalog = t.object database AND c.table schema = t.object_schema AND c.table_name = t.object_name AND c.column_name = t.column_name WHERE t.tag_name = 'PII' AND t.object_domain = 'COLUMN';
```

B.

```
SELECT table_catalog, table_schema, table_name, column_name, tag_name, tag_value FROM snowflake.account_usage.tag_references WHERE tag_name = 'PII' AND object_domain = 'COLUMN';
```

C.

```
SELECT t.table_name, t.column_name, tags.tag_name, tags.tag_value FROM snowflake.account_usage.tables t, TABLE(INFORMATION_SCHEMA.TAG_REFERENCES_ALL_COLUMNS('PII')) tags;
```

D.

```
SELECT OBJECT_DATABASE, OBJECT_SCHEMA, OBJECT_NAME, COLUMN_NAME, TAG_NAME, TAG_VALUE FROM snowflake.account_usage.tag_references WHERE TAG_NAME ='PII' and object_domain = 'COLUMN';
```

E.

```
SELECT OBJECT_DATABASE, OBJECT_SCHEMA, OBJECT_NAME, COLUMN_NAME, TAG_NAME, TAG_VALUE FROM snowflake.account_usage.tag_references WHERE TAG_NAME ='PII';
```

Answer: D

Explanation:

Option D provides the MOST comprehensive and accurate report. It directly queries the view, filtering for 'TAG_NAME = 'PII" and 'object_domain = 'COLUMN" to specifically target tags applied to columns. It selects the database, schema, table name, column name, tag name, and tag value, providing all the necessary information. Option A requires a JOIN between "snowflake.account_usage.columns' and , which is unnecessary for this use case and less efficient. option B is missing the OBJECT_DATABASE and OBJECT_SCHEMA which is needed to fully qualify the table. option C attempts to use a table

function, which is unnecessary complexity and potentially less performant. Option E does not filter for column-level tags, potentially including tags applied to other object types (e.g., tables, views), leading to inaccurate results. The fully qualified name can be easily constructed from OBJECT DATABASE, OBJECT SCHEMA and OBJECT NAME.