Snowflake
SOL-C01
SnowPro Associate: Platform Certification Exam

For More Information – Visit link below:
https://www.examsempire.com/

Product Version

1. **Up to Date products, reliable and verified.**
2. **Questions and Answers in PDF Format.**

Latest Version: 6.0

Question: 1

A data scientist is using Snowflake Notebooks to analyze sales data. They have encountered a situation where a specific cell containing a complex UDF that calculates customer lifetime value is consistently failing with an obscure error. The error message is not providing enough information to pinpoint the root cause. Considering the limitations and features of Snowflake Notebooks, what is the MOST effective approach to debug this UDF without disrupting the entire notebook session and while preserving the data context within the notebook environment?

A. Replace the UDF cell with a simple SELECT statement to confirm connectivity and basic notebook functionality, then gradually reintroduce complexity into the UDF code to isolate the issue.
B. Download the entire notebook as a Python script, run it in a local Python environment with enhanced debugging tools (e.g., pdb), and then translate the fixes back to the Snowflake Notebook.
C. Leverage the Snowflake query history to examine the compiled SQL generated by the UDF, searching for potential syntax errors or misinterpretations of input data types. Utilize EXPLAIN PLAN to understand query execution.
D. Isolate the UDF code and relevant input data into a separate Snowflake Stored Procedure. Debug the Stored Procedure using Snowflake's built-in debugging features, then reintegrate the corrected code into the notebook.
E. Since Snowflake Notebooks have limited debugging capabilities, the best approach is to rewrite the entire UDF using simpler SQL statements to avoid complex logic and potential errors.

Answer: D

Explanation:
Isolating the IJDF into a stored procedure allows for focused debugging using Snowflake's debugging tools. While options A and C are helpful for initial troubleshooting, they don't provide the same level of detailed debugging as option D. Option B is less efficient due to the need to translate code between environments, and option E might not be feasible if the UDF's complexity is necessary for accurate results.

Question: 2

A Snowflake Notebook is being used to train a machine learning model. The notebook contains multiple cells: one to load data, one to perform feature engineering, and one to train the model using Snowpark ML. After running the feature engineering cell, the notebook session unexpectedly terminates. No error messages are displayed, and the notebook kernel restarts automatically. The data being processed is approximately 50 GB. What are the TWO most likely causes of this issue and their corresponding remedies?

A. The notebook session timed out due to inactivity. Increase the SESSION_IDLE_TIMEOUT setting for the Snowflake user or account.

B. The Snowflake warehouse being used for the notebook session is undersized, leading to memory exhaustion and session termination. Scale up the warehouse to a larger size (e.g., from X-Small to Small or Medium).
C. The Snowpark ML library encountered an incompatibility issue with the current Snowflake version. Upgrade or downgrade the Snowpark ML version to match the compatible Snowflake version.
D. The feature engineering process resulted in an excessively large intermediate dataset that exceeded available memory. Optimize the feature engineering code to reduce memory consumption or partition the data.
E. The notebook is configured to automatically save checkpoints. This feature is causing overhead and instability when dealing with large datasets. Disable auto- saving in the notebook settings.

Answer: B,D

Explanation:
Warehouse size (B) is crucial for processing large datasets. An undersized warehouse will quickly run out of memory, leading to session termination. Memory exhaustion (D) from feature engineering is also a common cause. Optimizing the code and partitioning data can help. While session timeouts (A) are possible, the immediate termination after running the cell suggests a memory issue. Snowpark ML incompatibility (C) is less likely if the basic loading and processing work. Auto-saving (E) contributes to overhead but is unlikely to be the primary cause of immediate termination.

Question: 3

You are using a Snowflake Notebook to perform data transformations. You have a cell that defines a Snowpark DataFrame named 'df_sales'. You want to persist this DataFrame as a Snowflake table called ‘SALES TRANSFORMED' in a schema named 'ANALYTICS’. Which of the following code snippets will correctly achieve this, assuming the necessary imports and session setup are already completed?

A.
```
df_sales.write.saveAsTable("ANALYTICS.SALES_TRANSFORMED")
```
B.
```
df_sales.write.mode("overwrite").save_as_table("ANALYTICS.SALES_TRANSFORMED")
```
C.
```
df_sales.write.option("table", "ANALYTICS.SALES_TRANSFORMED").save()
```
D.
```
df_sales.write_pandas(table_name='ANALYTICS.SALES_TRANSFORMED', auto_create_table=True, overwrite=True)
```
E.
```
df_sales.createOrReplaceTempView("temp_sales")
session.sql("CREATE OR REPLACE TABLE ANALYTICS.SALES_TRANSFORMED AS SELECT   FROM temp_sales")
```

Answer: E

Explanation:
Option E uses the correct approach of creating a temporary view from the Snowpark DataFrame and then using a SQL CREATE OR REPLACE TABLE statement to persist the data. This method ensures the

data is correctly written to the specified schema and table. Option B is closer, but 'save_as_table' is not a function in the snowpark dataframe writer class. Option D is used when you have Pandas Dataframe to start with.

Question: 4

You're using a Snowflake Notebook to collaboratively develop a data pipeline. Several team members are working on the same notebook concurrently. One team member accidentally deletes a cell containing critical data transformation logic. What Snowflake features, accessible within or through the Notebook environment, can be used to recover the lost cell's content and minimize disruption to the workflow? Select TWO correct answers.

A. Use the 'Undo' function within the Snowflake Notebook editor to revert the deletion. If the 'Undo' history is insufficient, manually re-enter the code from memory or documentation.
B. Leverage the Snowflake Time Travel feature on the underlying table(s) used in the deleted cell's logic to retrieve the data at a point in time before the deletion occurred, then recreate the cell with the recovered data transformation logic.
C. Check the Snowflake Notebook's version history (if enabled) to revert to a previous version of the notebook before the cell was deleted.
D. Examine the Snowflake query history associated with the notebook session. The SQL statements executed in the deleted cell may be present in the query history, allowing you to reconstruct the code.
E. Since Snowflake Notebooks automatically back up cell contents to a secure cloud storage location, contact Snowflake support to request a restoration of the deleted cell.

Answer: C,D

Explanation:
Snowflake Notebook version history (C) is the primary mechanism for reverting to a previous state. The Snowflake query history (D) will contain the SQL commands executed by the cell, allowing for reconstruction of the logic. 'Undo' (A) is limited and may not be sufficient. Time Travel (B) applies to data , not code in the notebook. Snowflake does not automatically back up individual cells for restoration (E).

Question: 5

A Snowflake Notebook is configured to run a series of data processing steps on a schedule using the Snowflake Task feature. One of the steps involves calling an external API to enrich the data. Due to network instability, the API calls occasionally fail, causing the entire task to fail. You want to implement error handling within the notebook to retry failed API calls up to a certain number of times before giving up. Assuming you are using Python within the Snowflake Notebook and Snowpark, how would you best implement this retry mechanism while ensuring minimal disruption to the notebook's workflow and preserving error information?

A. Wrap the API call within a 'try...except' block. Inside the 'except' block, use 'time.sleep()' to introduce a delay, and then recursively call the function containing the API call until the maximum number of retries is reached. If all retries fail, log the error and re-raise the exception.

B. Implement a custom decorator function that handles retries. The decorator should catch exceptions from the API call, log the error, introduce a delay, and retry the API call until the maximum number of attempts is reached. Apply this decorator to the function containing the API call.
C. Use the ‘snowflake.snowpark.functions.call_udf function with the 'retry_count' parameter set to the desired number of retries. This will automatically handle retries for IJDF calls within the notebook.
D. Utilize the Snowflake Task's built-in error handling capabilities to automatically retry the task upon failure. Configure the task to retry a specific number of times with a specified delay between retries.
E. Use a dedicated Python library like 'tenacity' within the Snowflake Notebook. Configure ‘tenacity' to retry the API call with exponential backoff and custom exception handling. Log all retry attempts and final error messages.

Answer: E

Explanation:
Using a dedicated library like ‘tenacity' (E) provides the most robust and flexible retry mechanism. It offers features like exponential backoff, customizable retry strategies, and detailed logging. While ‘try...except’ blocks (A) can work, they are less elegant and harder to maintain for complex retry logic. Snowflake does not have direct error handling mechanisms for external API calls (C, D) within notebooks. A decorator function (B) can work, but ‘tenacity’ provides more features with less code.

Question: 6

You are developing a Snowflake Notebook to analyze sales data. You want to create a dynamic SQL query that filters data based on a parameter passed from a user interface element (e.g., a dropdown). How can you best achieve this within a Snowflake Notebook, ensuring SQL injection vulnerabilities are mitigated?

A. Directly concatenate the parameter value into the SQL query string using Python string formatting.
B. Use Snowflake's built-in parameterization feature within the SQL cell. For example: ‘‘‘sql SELECT FROM sales WHERE region = :region_param; and then assign the ‘region_param’ variable in a Python cell.
C. Use the ‘snowflake.connector’ library to create a prepared statement with parameterized query execution from a Python cell.
D. Store the parameter in a Snowflake session variable and reference the session variable in the SQL query string.
E. Use JavaScript stored procedure to generate SQL string and execute the query.

Answer: B,C

Explanation:
Options B and C are the most secure and recommended approaches. Option B leverages Snowflake Notebook's built-in parameterization, which handles escaping and prevents SQL injection. Option C uses the ‘snowflake.connector’ to achieve similar result. Option A is highly susceptible to SQL injection. Option D is less suitable for dynamic parameters from a I-Jl. Option E increases the complexity significantly and doesn't directly address parameterization within the notebook environment.

Question: 7

You have a Snowflake Notebook that performs several complex data transformations. The notebook is taking longer than expected to run, and you need to optimize its performance. Which of the following actions would likely improve the notebook's execution speed the most?

A. Upgrade the Snowflake virtual warehouse to a larger size while the notebook is running.
B. Re-order the notebook cells so that SQL cells are grouped together and Python cells are grouped together.
C. Use the 'LIMIT clause in your SQL queries to only process a subset of the data during development and testing.
D. Explicitly commit transactions after each SQL cell execution.
E. Optimize SQL queries for performance using techniques such as clustering, partitioning, and appropriate join strategies.

Answer: A,E

Explanation:
Options A and E are correct. Option A: Upgrading the warehouse size provides more resources (CPU, memory) for the notebook's computations. Option E: Optimized SQL queries are fundamental for performance in Snowflake; correct use of clustering, partitioning and efficient joins drastically reduces execution time. Option B has a minimal impact. Option C is only useful during development, not for production performance. Option D is unnecessary in most cases as Snowflake typically autocommits; explicitly committing will likely degrade performance, not improve it.

Question: 8

In a Snowflake Notebook, you're attempting to read data from a Snowflake table named 'CUSTOMER DATA into a Pandas DataFrame for further analysis. The table contains a column named of data type TIMESTAMP NTZ. Which of the following Python code snippets will successfully read the data and preserve the 'CREATED_AT column as a datetime object in the DataFrame?

- ☐ ```python import snowflake.connector import pandas as pd ctx = snowflake.connector.connect(...) cs = ctx.cursor() cs.execute("SELECT FROM CUSTOMER_DATA") df = cs.fetch_pandas_all() ```
- ☐ ```python import snowflake.connector import pandas as pd ctx = snowflake.connector.connect(...) cs = ctx.cursor() cs.execute("SELECT FROM CUSTOMER_DATA") df = pd.DataFrame(cs.fetchall(), columns=[col[0] for col in cs.description]) df['CREATED_AT'] = pd.to_datetime(df['CREATED_AT']) ```
- ☐ ```python import snowflake.connector import pandas as pd ctx = snowflake.connector.connect(...) cs = ctx.cursor() cs.execute("SELECT FROM CUSTOMER_DATA") df = cs.fetch_pandas_all() df['CREATED_AT'] = df['CREATED_AT'].astype('datetime64[ns]') ```
- ☐ ```python import snowflake.connector import pandas as pd ctx = snowflake.connector.connect(...) cs = ctx.cursor() sql = "SELECT FROM CUSTOMER_DATA" df = pd.read_sql(sql, ctx) ```
- ☐ ```python import snowflake.connector import pandas as pd ctx = snowflake.connector.connect(...) cs = ctx.cursor() cs.execute("SELECT CAST(CREATED_AT AS STRING), FROM CUSTOMER_DATA") df = cs.fetch_pandas_all() ```

A. Option A
B. Option B
C. Option C
D. Option D

E. Option E

Answer: A,D

Explanation:
Options A and D are the most efficient and correct. automatically handles the conversion of Snowflake TIMESTAMP NTZ to Pandas datetime objects. Option D 'pd.read_sqr using the connection context is also good , which handles the conversion automatically as well. Option B fetches the data as tuples, requiring manual column naming and datetime conversion, and it is inefficient. Option C requires to work correctly. Option E cast timestamp column explicitly and defeats purpose.

Question: 9

You are working with a Snowflake Notebook to process data from an external stage (AWS S3). You need to access the S3 stage using a named stage object and a storage integration configured with IAM roles. Which of the following options represents the correct sequence of steps and Snowflake SQL commands within the notebook to achieve this?

A. 1. Create the storage integration with appropriate IAM roles. 2. Create the external stage referencing the storage integration. 3. Use 'COPY INTO' command to load data from the stage into a Snowflake table, specifying the stage name.
B. 1. Create the external stage specifying the S3 bucket URL and credentials. 2. Create a file format object. 3. Use 'SELECT FROM @stage/file.csv' to query the data directly from the stage.
C. 1. Create the storage integration with appropriate IAM roles. 2. Use the 'LIST @stage' command to verify the stage connectivity and file listing. 3. Create an external table pointing to the external stage. 4. Use the 'REFRESH EXTERNAL TABLE command to load the metadata. 5. Query data directly from the external table.
D. 1. Create the storage integration with appropriate IAM roles. 2. Create the external stage referencing the storage integration. 3. Create a file format object. 4. Use 'COPY INTO command to load data from the stage into a Snowflake table, specifying the stage name and the file format.
E. 1. Create the external stage specifying the S3 bucket URL and credentials. 2. Create a file format object. 3. Use 'COPY INTO' command to load data from the stage into a Snowflake table, specifying the stage name and the file format.

Answer: A,D

Explanation:
Options A and D are correct. The correct and secure approach involves using storage integrations with IAM roles. Creating the storage integration first establishes trust between Snowflake and AWS. The stage then references this integration. The 'COPY INTO' command (Option A) is used for loading data into Snowflake tables. External tables (option C) are read-only and designed for querying data in place, not loading it. Option D Corrects A by including the file format, which is generally required. Option B does not use storage integrations, which is more secure . Option E Requires credential in stage definition, which is not encouraged.

Question: 10

You're building a Snowflake Notebook to automate data quality checks on a daily basis. You have a series of SQL queries, each representing a different quality rule (e.g., checking for null values, duplicate records, invalid date formats). You want to implement error handling so that if one quality check fails, the notebook continues to execute the remaining checks and logs the errors. Which is the most robust approach to achieve this within the Snowflake Notebook environment, minimizing code complexity and maximizing fault tolerance?

A. Wrap each SQL query execution in a Python 'try...except' block and log any exceptions to a Python list. After all checks are completed, print the list of errors.
B. Use Snowflake's 'SYSTEM$LAST_CHANGE COMMIT _ TIME function to check if the SQL query executed successfully after each check. If not, log the error.
C. Create a stored procedure in Snowflake that encapsulates each quality check and handles its own error logging. Call the stored procedures sequentially from the notebook.
D. Within each SQL query, use Snowflake's ' TRY_CAST or similar error-handling functions to handle individual row errors and then aggregate the error counts at the end of the query.
E. Enable the 'AUTO_RETRY parameter at the account level, so failed queries are automatically retried after 5 seconds.

Answer: A,C

Explanation:
Options A and C represent the most appropriate solutions. Option A provides explicit error handling in Python, allowing the notebook to continue execution even if some SQL queries fail. It is simple to implement and offers good control over error logging. Option C encapsulates the checks in stored procedures within Snowflake. This allows for modularity and allows Snowflake to handle transaction management (e.g. error handling) for each data quality check individually, improving fault tolerance. It also allows for easier reuse and maintainability. Option B doesn't provide a direct way to determine if the query succeeded in returning the correct result; it only checks for a commit time. Option D is relevant for handling errors within the data itself, but doesn't prevent a query from failing entirely. Option E doesn't log/handle the errors; it only retries them.

Question: 11

You are developing a Snowflake Notebook to analyze sales data. You need to connect to a Snowflake database using Python and execute a query to retrieve the top 10 products by sales volume. Which of the following code snippets is the MOST efficient and secure way to achieve this, assuming you've already configured the necessary connection details and have 'snowflake.connector' installed?

A.
```
import snowflake.connector ctx = snowflake.connector.connect( user='', password='', account='' ) cs = ctx.cursor() cs.execute("SELECT product_name, SUM(quantity) AS total_quantity FROM sales GROUP BY product_name ORDER BY total_quantity DESC LIMIT 10") for (product_name, total_quantity) in cs: print(f'{product_name}: {total_quantity}') ctx.close()
```
B.
```
import snowflake.connector ctx = snowflake.connector.connect( user=os.environ['SNOWFLAKE_USER'], password=os.environ['SNOWFLAKE_PASSWORD'], account=os.environ['SNOWFLAKE_ACCOUNT'] ) cs = ctx.cursor() query = "SELECT product_name, SUM(quantity) AS total_quantity FROM sales GROUP BY product_name ORDER BY total_quantity DESC LIMIT 10" cs.execute(query) results = cs.fetchall() for row in results: print(row) ctx.close()
```
C.

```
import snowflake.connector import os ctx = snowflake.connector.connect( user=os.environ['SNOWFLAKE_USER'], password=os.environ['SNOWFLAKE_PASSWORD'], account=os.environ['SNOWFLAKE_ACCOUNT'], warehouse=os.environ.get('SNOWFLAKE_WAREHOUSE', 'COMPUTE_WH'), database=os.environ.get('SNOWFLAKE_DATABASE', 'SALES_DB'), schema=os.environ.get('SNOWFLAKE_SCHEMA', 'PUBLIC') ) cs = ctx.cursor() query = "SELECT product_name, SUM(quantity) AS total_quantity FROM sales GROUP BY product_name ORDER BY total_quantity DESC LIMIT 10" cs.execute(query) for product_name, total_quantity in cs.fetchall(): print(f'{product_name}: {total_quantity}') ctx.close()
```

D.

```
import snowflake.connector ctx = snowflake.connector.connect( user='', password='', account='' ) cs = ctx.cursor() query = "SELECT product_name, SUM(quantity) AS total_quantity FROM sales GROUP BY product_name ORDER BY total_quantity DESC LIMIT 10" cs.execute(query) results = cs.fetchmany(10) for row in results: print(row) ctx.close()
```

E.

```
import snowflake.connector import os ctx = snowflake.connector.connect( user=os.environ['SNOWFLAKE_USER'], password=os.environ['SNOWFLAKE_PASSWORD'], account=os.environ['SNOWFLAKE_ACCOUNT'] ) cs = ctx.cursor() query = "SELECT product_name, SUM(quantity) AS total_quantity FROM sales GROUP BY product_name ORDER BY total_quantity DESC LIMIT 10" try: cs.execute(query) results = cs.fetchall() for row in results: print(row) except snowflake.connector.errors.ProgrammingError as e: print(f"Error executing query: {e}") finally: cs.close() ctx.close()
```

Answer: E

Explanation:
Option E is the most secure and robust. It utilizes environment variables to store credentials securely, includes error handling for potential query execution issues, and ensures that the cursor and connection are closed properly in a 'finally' block, even if an error occurs. Option A hardcodes credentials, which is a major security risk. Options B and C don't include comprehensive error handling. Option D uses 'fetchmany(10)' , which might not fetch all results if there are exactly 10 rows and is less common for retrieving all rows.

Question: 12

Within a Snowflake Notebook, you have a Python script that performs several data transformations using Pandas DataFrames and then attempts to load the transformed data into a new Snowflake table. The script fails intermittently with 'MemoryError'. Which of the following strategies could you employ to mitigate the 'MemoryError' when working with Snowflake Notebooks and Python? (Choose all that apply.)

A. Increase the virtual warehouse size allocated to the Snowflake session.
B. Implement chunking: Read the data from Snowflake in smaller batches, process each batch individually, and append the results to the target table.
C. Optimize the Pandas DataFrame's data types to use less memory (e.g., using 'int32 instead of 'int64' where appropriate).
D. Utilize the 'COPY INTO' command directly from Snowflake to load data into the table, bypassing Pandas DataFrames entirely.
E. Reduce the number of concurrent Snowflake Notebook sessions running.

Answer: B,C,D

Explanation:
Options B, C, and D are correct. A MemoryError indicates that the Python process within the Snowflake Notebook is running out of memory. Chunking (B) allows processing of data in smaller, manageable pieces. Optimizing Pandas DataFrame data types (C) reduces the memory footprint of each DataFrame. Using 'COPY INTO' (D) leverages Snowflake's internal data loading capabilities, avoiding the need to load

the entire dataset into a Pandas DataFrame. Increasing the virtual warehouse size (A) primarily affects query performance within Snowflake, not the memory available to the Python process in the Notebook. Reducing the number of concurrent sessions might help if resource contention is the root cause, but is less directly related to the error, so E is not the strongest solution.

Thank You for Trying Our Product

Special 16 USD Discount Coupon: NSZUBG3X

Email: support@examsempire.com

Check our Customer Testimonials and ratings available on every product page.

Visit our website.

https://examsempire.com/