Snowflake

SPS-C01

SnowPro Specialty: Snowpark 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 engineering team is using Snowpark Python to build a complex ETL pipeline. They notice that certain transformations are not being executed despite being defined in the code. Which of the following are potential reasons why transformations in Snowpark might not be executed immediately, reflecting the principle of lazy evaluation? Select TWO correct answers.

A. Snowpark operations are only executed when an action (e.g., ‘collect()', ‘show()', is called on the DataFrame or when the DataFrame is materialized.
B. The ‘eager_execution’ session parameter is set to 'True'.
C. Snowpark automatically executes all transformations as soon as they are defined, regardless of whether the results are needed.
D. Snowpark employs lazy evaluation to optimize query execution by delaying the execution of transformations until the results are actually required.
E. The size of the data being processed exceeds Snowflake's memory limits, causing transformations to be skipped.

Answer: A,D

Explanation:
Snowpark employs lazy evaluation, which means transformations are not executed until an action is performed on the DataFrame. This allows Snowflake to optimize the entire query plan before execution. Setting ‘eager_execution’ to True does NOT exist in Snowpark Python. Data size exceeding Snowflake's limits would result in an error, not skipped transformations.

Question: 2

You are developing a Snowpark Python application that reads data from a Snowflake table, performs several transformations including filtering, aggregation, and joining with another DataFrame, and then writes the results back to a new table. You want to optimize the execution plan to minimize data movement and processing time. Which of the following strategies would be MOST effective in leveraging Snowpark's lazy evaluation capabilities to achieve this optimization?

A. Calling ‘cache()' on the initial DataFrame read from the table to materialize it in memory before any transformations.
B. Defining all transformations in a single, complex SQL query string and using to execute it.
C. Chaining all the transformations together using DataFrame methods (e.g., 'filter()' , ‘groupBy()' , ‘join()') and only calling or at the very end.
D. Calling after each transformation to materialize intermediate results and then creating new DataFrames for subsequent operations.
E. Executing each transformation in separate Python processes using multiprocessing to parallelize the workload.

Answer: C

Explanation:
Chaining transformations and delaying execution until the final action allows Snowpark to optimize the entire query plan. Caching the initial DataFrame might improve performance in some cases, but it can also introduce unnecessary materialization. Defining transformations in a single SQL query string bypasses Snowpark's optimization capabilities. Calling 'collect()' after each transformation defeats the purpose of lazy evaluation. Python multiprocessing does not directly interact with Snowpark's query optimization.

Question: 3

Consider the following Snowpark Python code snippet:

A. The final 'DataFrame' 'df3' will only contain rows where 'coll' is greater than 10 and 'c012 is not null.
B. The code will result in a compilation error because 'df2 is not explicitly materialized before being used.
C. The code will execute without error, and Snowflake's query optimizer will likely combine the two 'filter' operations into a single scan of the table.
D. Two separate scans of the 'my_table' table will always be performed, one for each 'filter' operation.
E. The code will throw a NullPointerException as null values are not allowed in Snowpark DataFrames.

Answer: C

Explanation:
Snowpark's lazy evaluation allows the query optimizer to combine multiple filter operations into a single scan, improving efficiency. The code will execute successfully because Snowpark does not require explicit materialization of intermediate DataFrames. Null values are allowed in Snowpark DataFrames.

Question: 4

You are building a Snowpark application that involves a UDF. Consider that you are creating UDF as follows:

A. UDF registration happens on the client side, before dataframe transformations are evaluated and sent to Snowflake.
B. UDF registration occurs on the server-side, only when a dataframe action such as collect or show is called .
C. The UDF is registered lazily; only when called the first time will it be registered and available in the session.
D. There will be no attempt to register UDF until .collect()' action is performed.
E. The behaviour of the UDF registration dependends on the size of code in it.

Answer: A

Explanation:
UDFs are registered client-side when created which happens before any dataframe is sent for processing. Lazy Evaluation applies to dataframe transformations and not to registration of UDFs.

Question: 5

You are tasked with building a Snowpark application to perform sentiment analysis on customer reviews stored in a Snowflake table named 'CUSTOMER REVIEWS'. The application should be deployed as a UDF. The sentiment analysis is performed by a third-party Python library, 'sentiment_analyzer'. Due to security constraints, direct internet access is prohibited from within the Snowflake environment. What steps are necessary to ensure the 'sentiment_analyzer' library can be used by your Snowpark UDF?

A. Package the ‘sentiment_analyzer’ library into a JAR file and upload it to a Snowflake stage, then specify the JAR in the 'imports' parameter of the UDF creation statement.
B. Package the ‘sentiment_analyzer’ library into a ZIP file and upload it to a Snowflake stage, then specify the ZIP in the 'imports' parameter of the UDF creation statement.
C. Install the ‘sentiment_analyzer’ library using ‘conda install' directly within the Snowpark session before creating the UDF.
D. Request Snowflake support to whitelist the ‘sentiment_analyzer’ library for direct download during UDF execution.
E. Use the ‘packages' parameter in the UDF creation statement to specify the ‘sentiment_analyzer’ library from Anaconda.

Answer: E

Explanation:
The ‘packages' parameter in the UDF creation statement allows specifying Python packages from the Anaconda repository, which are then automatically made available to the UDF during execution. This is the recommended approach when direct internet access is restricted. Options A and B are incorrect because these steps would be used to include a Java library, not a Python library. Option C is incorrect because you cannot directly install packages within a Snowpark session in this way. Option D is not a standard procedure.

Question: 6

Consider the following Snowpark Python code snippet:

A.

```
import snowflake.snowpark as snowpark
from snowflake.snowpark.functions import col

def main(session: snowpark.Session):
    df = session.table('MY_TABLE')
    result_df = df.groupBy(col('CATEGORY')).agg(snowpark.functions.sum(col('SALES')).alias('TOTAL_SALES'))
    result_df.write.mode('overwrite').save_as_table('AGGREGATED_SALES')
    return 'Aggregation complete'
```

Given this code, which of the following statements are correct?
B. This code will ovemrite the table if it already exists.
C. The ‘result_df DataFrame will be persisted to the 'AGGREGATED SALES table in the default schema of the user running the code.
D. The code will fail because there is no call to or on the ‘result_df dataframe and Snowflake performs lazy evaluation.
E. This code will fail because is not a valid method for Snowpark DataFrames.

Answer: B,C

Explanation:
`write.mode('overwrite').save_as_table('AGGREGATED_SALES')` ' will indeed overwrite the table if it exists. When a schema is not explicitly specified, Snowpark defaults to the schema of the user's current session for table creation. No explicit call to an action like 'collect()' or show()' is needed to trigger execution, as is an action itself. is a valid operation to persist DataFrames as tables in Snowpark. The table name does not strictly need to be fully qualified unless targeting a schema different from the user's default.

Question: 7

You have a Snowpark DataFrame named 'orders df with columns 'order id', 'customer id', 'order date', and 'total amount'. You need to create a new DataFrame that contains only the 'customer_id" and the total number of orders placed by each customer. However, you want to perform this aggregation in parallel using a user-defined function (UDF) to improve performance. Which approach is MOST efficient and CORRECT?

A. Create a UDF that takes a customer ID as input and returns the count of orders for that customer, then apply this UDF to each distinct customer ID using ‘map'.
B. Create a UDF that performs the entire aggregation and call it with ‘orders_df. The UDF uses a Pandas DataFrame internally to perform the count.
C. Use ‘groupBy' to group by 'customer_id' and then use the 'count' aggregate function. Do not use a UDF.
D. Create a UDF that takes an iterator of order IDs and returns the count. Apply this UDF to each partition of the DataFrame using ‘maplnPartitions’ .
E. Create a UDF that calculates the mode of total_amount, then apply the UDF to the DataFrame using ‘select'.

Answer: C

Explanation:
The most efficient and correct approach is to use the built-in ‘groupBy’ and ‘count' functions. These functions are optimized for Snowflake's architecture and will generally outperform UDF-based solutions for simple aggregations. Using UDFs for simple tasks introduces overhead and can negate any potential performance benefits. While options A, B and D could achieve the result, they are less efficient. Option E is irrelevant to the problem.

Question: 8

You are working with a Snowpark DataFrame representing sensor data. The DataFrame contains columns like 'timestamp', 'sensor id' , and 'value'. You need to perform a complex windowing operation to calculate the moving average of the 'value' for each 'sensor id' over a 5-minute window, but only for data points where the 'value' is greater than a threshold. The window should be defined based on the 'timestamp' column. What is the most efficient and correct approach to implement this using Snowpark DataFrames?

A. Use a combination of 'filter' to apply the threshold condition, 'Window.partitionBy' and 'Window.orderBy' to define the window, and 'avg' window function to calculate the moving average.
B. First, collect the entire DataFrame into a Pandas DataFrame, then use Pandas windowing functions to calculate the moving average.
C. Create a UDF that takes a list of timestamps and values as input and returns the moving average. Apply this UDF to the entire DataFrame.
D. Use a loop to iterate over each 'sensor_id' , filter the DataFrame for that sensor, calculate the moving average using Pandas windowing functions, and then combine the results.
E. First apply the moving average calculation to the DataFrame and then filter for rows with values exceeding the threshold, since calculations are performed in order.

Answer: A

Explanation:
The most efficient and correct approach is to use Snowpark's built-in windowing functions. Applying the threshold using 'filter' before the windowing operation reduces the amount of data processed by the window function, improving performance. Using 'Window.partitionBy' and 'Window.orderBy' correctly defines the window based on 'sensor_id' and 'timestamp', respectively. Using 'avg' window function calculates the moving average within the defined window. Options B, C, and D are less efficient because they involve transferring data to the client side (Pandas) or using UDFs, which can introduce overhead. Option E reverses the correct process.

Question: 9

A data engineering team is using Snowpark Python to build a data pipeline. They need to create a User-Defined Function (UDF) that
transforms a JSON string column representing customer information into a STRUCT type containing flattened fields for 'name', 'age', and 'city'. The UDF should handle null values gracefully and return NULL if the input JSON is invalid or if the 'name' field is missing. Considering performance implications and error handling, which of the following approaches is MOST optimal for defining and registering this UDF?

A. Using 'snowflake.snowpark.functions.udf with and handling JSON parsing and field extraction using standard Python libraries within the UDF, returning a JSON string representation of the STRUCT.
B. Using 'snowflake.snowpark.functions.udf with defining the STRUCT schema explicitly, and handling JSON parsing and field extraction using the 'snowflake.snowpark.functions.parse_json' function. Return None for invalid json.

C. Using ‘session.register_function’ to register a Python function as a UDF with and manually constructing a VARIANT object in Python from the extracted JSON fields.
D. Using ‘snowflake.snowpark.functions.sproc’ to create a stored procedure that performs the JSON transformation and returns the transformed data.
E. Using ‘snowflake.snowpark.functions.udf with and relying solely on Snowflake's built-in JSON functions within the UDF, even for complex transformations, and handling exceptions with try-except blocks within the UDF to return NULL.

Answer: B

Explanation:
Option B is the most optimal. Using allows Snowpark to understand the schema of the returned data, enabling efficient type checking and query optimization. ‘snowflake.snowpark.functions.parse_json’ leverages Snowflake's internal JSON parsing capabilities, leading to better performance. Returning None from UDF handles nulls gracefully. Other options either involve less efficient StringType return types, manual VARIANT object creation which is less type-safe, or suggest stored procedures when a simple UDF is sufficient.

Question: 10

A Snowpark application needs to process large volumes of sensor data stored in a Snowflake table named , which includes columns , ‘timestamp’ , and The application must calculate a rolling average of for each over a 5-minute window. The data is not perfectly ordered by 'timestamp' within each 'sensor_id'. What is the MOST efficient and accurate way to implement this rolling average calculation using Snowpark?

A. Using after applying a filter to select only the data within the 5-minute window, updating the filter for each new window.
B. Using a Window specification with ‘orderBy('timestamp')' and ‘rowsBetween(Window.unboundedPreceding, Window.currentRow)' to calculate the cumulative average, then subtracting the average from 5 minutes ago. The query will then be grouped on the sensor id.
C. Using a Window specification with ‘orderBy('timestamp')' and ‘rowsBetween(Window.unboundedPreceding, Window.currentRow)' in conjunction with and a UDF to manually calculate the rolling average within each group.
D. Using a Window specification with 0)' and the 'avg()' window function. (Where ‘to_seconds’ converts a duration to seconds)
E. Implementing a Python UDTF (User-Defined Table Function) that iterates through the data for each calculates the rolling average manually, and emits the results as rows.

Answer: D

Explanation:
Option D is the most efficient and accurate. ‘partitionBy('sensor_id')' ensures that the rolling average is calculated separately for each sensor. ‘orderBy('timestamp'Y orders the data within each partition by timestamp. 0)' defines the 5- minute window relative to the current row, accurately capturing all

readings within that window even if they are slightly out of order. ‘avg(Y then efficiently calculates the average within that window. Other options are either less efficient (e.g., UDTF iteration) or less accurate (e.g., incorrect window definitions, filtering).

Question: 11

You are tasked with optimizing a Snowpark Python application that performs complex data transformations on a large dataset. The application is running slower than expected, and you suspect that data serialization and transfer between the Snowpark client and the Snowflake engine are bottlenecks. Which of the following strategies could you implement to improve performance? (Select all that apply.)

A. Minimize the amount of data transferred between the client and the engine by pushing down as much computation as possible to Snowflake using Snowpark DataFrame operations.
B. Utilize smaller batch sizes when writing data back to Snowflake to reduce memory pressure on the client.
C. Create and utilize temporary tables within Snowflake to store intermediate results of complex transformations.
D. Convert all dataframes to Pandas dataframes locally and perform data manipulation with Pandas methods to take advantage of local resources.
E. Increase the configuration parameter to maximize parallelism within the Snowpark engine without considering resources or potential bottleneck.

Answer: A,B,C

Explanation:
Options A, B, and C are correct strategies. Pushing down computation (A) reduces data transfer. Using smaller batch sizes (B) can reduce memory pressure, especially for large datasets. Using temporary tables (C) allows intermediate results to be stored and processed entirely within Snowflake, avoiding unnecessary data transfer. Option D is incorrect because converting to Pandas DataFrames brings the data to the client, negating the benefits of Snowpark's distributed processing. Option E is dangerous since it could cause bottleneck if the resources are not managed correctly.

Question: 12

You have a Snowpark Python application that interacts with Snowflake using a service account. You are rotating the private key associated with the service account. After updating the private key in your application's configuration, you encounter an error during the connection attempt: ‘SnowflakeSQLException: 390103 (OSAOO): Failed to connect to DB. Encountered exception while creating connection: Authentication token has expired.' What is the MOST likely cause of this error, and what steps should you take to resolve it?

A. The Snowflake cache still holds the old private key. Clear the Snowflake connection cache in the application by calling and restarting the application.
B. The Snowflake service account hasn't been granted sufficient permissions to access the required resources. Re-grant the necessary roles and privileges to the service account.

C. The public key associated with the new private key has not been authorized in Snowflake for the service account. Ensure that the public key is associated with the service account using ALTER SERVICE ACCOUNT SET RSA PUBLIC KEY =”;’
D. The private key is in an incorrect format. Ensure that the private key is in PKCS#8 format and is properly encoded.
E. The connection string contains invalid characters. Ensure the account identifier and other parameters are correctly specified.

Answer: C

Explanation:
The error 'Authentication token has expired' in the context of a service account and private key rotation strongly suggests that the Snowflake instance has not been updated with the new public key that corresponds to the updated private key. Snowflake uses the public key to verify the authenticity of the client using the private key. Option C directly addresses this: The public key must be updated in Snowflake using the 'ALTER SERVICE ACCOUNT command to match the new private key being used by the application. Option A, although potentially helpful in other scenarios, does not address the core issue of mismatched key pairs. Options B, D, and E address other potential problems but are less likely in this specific scenario where the error occurs after a key rotation.

Question: 13

You are tasked with building a Snowpark application to process sensor data from IoT devices. The data arrives as JSON strings and needs to be transformed into a tabular format before being stored in a Snowflake table. You decide to use a User-Defined Table Function (UDTF) written in Python to handle this transformation. Which of the following approaches is the MOST efficient and scalable way to deploy and execute this UDTF in a production Snowpark environment, considering the possibility of high data volumes and concurrency?

A. Deploy the UDTF as an inline Python function within the Snowpark DataFrame transformation code, relying on the client-side Python interpreter for execution.
B. Deploy the UDTF using the '@udtf decorator and explicitly specify a parameter when registering the function to control the size of input batches processed by each worker node.
C. Deploy the UDTF as a stored procedure that is then called from your Snowpark application. The UDTF processes the data serially, one record at a time.
D. Deploy the UDTF as an external function, leveraging an external API gateway to invoke the Python code running on a serverless compute platform.
E. Package the Python UDTF into a zip file and upload it to a Snowflake stage. During UDTF registration, specify the stage location and use [...l, return_type=..., packages=['snowflake-snowpark-python', 'your_package'])'. Ensure that is appropriately tuned based on data size and complexity.

Answer: E

Explanation:
Option E is the MOST efficient and scalable. Deploying the UDTF with explicit package management and optimized batch sizing allows Snowflake to distribute the processing across multiple worker nodes,

leveraging the platform's parallel processing capabilities. Specifying dependencies using the ‘packages’ parameter ensures that the required Python libraries are available on each worker node. Properly tuning max_batch_size’ prevents memory exhaustion and optimizes processing speed. Options A, C, and D are generally less scalable and efficient due to client-side execution, serial processing, or added external latency. Option B is correct in mentioning batch sizing, but is incomplete. Option E offers the best balance of performance, scalability, and maintainability.

Question: 14

You have a Snowpark Python application that reads data from a Snowflake table named 'SALES DATA', performs several transformations using DataFrames, and then writes the results back to a new table named 'AGGREGATED SALES'. The application runs successfully, but you notice that the write operation to 'AGGREGATED SALES' is consistently slow. After examining the query profile, you observe significant skew in the data being written, causing some worker nodes to be overloaded. Which of the following techniques could you use within your Snowpark application to mitigate the data skew and improve the write performance to 'AGGREGATED SALES'?

A. Increase the size of the Snowflake warehouse being used to execute the Snowpark application. This will provide more compute resources to handle the data skew.
B. Use the method to evenly redistribute the data across a larger number of partitions before writing it to 'AGGREGATED SALES'.
C. Use the 'DataFrame.sort(col)' method to sort the data by the skew key before writing it to 'AGGREGATED SALES'. This will ensure that rows with similar values are processed by the same worker node.
D. Implement custom partitioning logic using a User-Defined Function (UDF) that calculates a hash value based on the skew key and then uses the 'DataFrame.repartitionByRange(col)' method to partition the data based on the hash values.
E. Use the method to specify a clustering key on the 'AGGREGATED SALES' table during table creation. This will physically organize the data on disk based on the skew key, improving write performance.

Answer: B,D

Explanation:
Both options B and D address data skew directly. Option B, , attempts to redistribute data evenly, which can alleviate skew if the repartitioning strategy is effective (e.g., using a hash function). Option D, using a UDF and 'repartitionByRange’ , allows for more sophisticated custom partitioning based on the skew key, potentially achieving a more balanced distribution. Increasing warehouse size (A) might provide more resources, but it doesn't directly address the skew. Sorting (C) can exacerbate skew by concentrating similar values on single nodes. Clustering (E) improves read performance after the data is written, but does not improve the write performance itself. Therefore, B and D are the best choices to reduce skew during the write operation.

Question: 15

Consider the following Snowpark Python code snippet that defines and registers a User-Defined Table Function (UDTF):

```
import snowflake.snowpark.functions as F
from snowflake.snowpark.types import IntegerType, StringType, StructType, StructField
from snowflake.snowpark import Row

@F.udtf(output_schema=StructType([StructField("id", IntegerType()), StructField("name", StringType())]), input_types=[StringType()])
class MyUDTF:
    def process(self, input_string: str):
        for i, word in enumerate(input_string.split()):
            yield Row(i, word)
```

Which of the following statements is MOST accurate regarding the behavior and limitations of this UDTF when used in a Snowpark DataFrame transformation?

A. The UDTF will process each input string in parallel, with Snowflake automatically distributing the processing across multiple worker nodes.
B. The UDTF can only be used with DataFrames that have been explicitly persisted as Snowflake tables.
C. The 'input_string' argument passed to the 'process' method will always be a single string value, even if the input DataFrame column contains NULL values.
D. The UDTF will be executed within the same Python process as the Snowpark driver program, limiting its scalability for large datasets.
E. If the input DataFrame column contains NULL values, the 'process' method will receive 'None' as the value for 'input_string'. The 'output_schema' correctly defines the structure of the output rows.

Answer: E

Explanation:
Option E is the most accurate. When a Snowpark UDTF receives NULL as input, it's passed as 'None' in Python. The provided code defines the 'output_schema' which describes the structure and types of the rows that the UDTF will return. Option A is incorrect because, while Snowflake distributes UDTF processing, the code itself doesn't guarantee parallelism within a single input string. Option B is incorrect; UDTFs can be used with any DataFrame, regardless of whether it's backed by a persistent table. Option C is incorrect because NULL values in the input DataFrame will be passed as 'None' to the 'process' method. Option D is incorrect; Snowpark distributes UDTF execution across worker nodes, not within the driver process.

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/