

Snowflake.NAS-C01.v2026-09-16.q161

Exam Code:	NAS-C01
Exam Name:	SnowPro Specialty - Native Apps
Certification Provider:	Snowflake
Free Question Number:	161
Version:	v2026-09-16
# of views:	106
# of Questions views:	1728
https://www.freecram.net/torrent/Snowflake.NAS-C01.v2026-09-16.q161.html	

NEW QUESTION: 1

A Snowflake Native App includes a managed task 'app_db.app_schema.daily_data_processing'. You, as the application provider, need to ensure that consumer accounts can only execute this managed task and have no other privileges. You also need to implement robust error handling to log any failed execution attempts within the consumer account. Which of the following SQL statements, when executed within the application's installation script, provides the most secure and complete solution to achieve these requirements, including error logging within the consumer account (assuming an existing table exists)?

A. GRANT EXECUTE MANAGED TASK ON MANAGED TASK TO APPLICATION ROLE app_public;

```
BEGIN
  GRANT EXECUTE MANAGED TASK ON MANAGED TASK app_db.app_schema.daily_data_processing TO APPLICATION ROLE app_public;
EXCEPTION
  WHEN OTHER THEN
    INSERT INTO app_db.app_schema.execution_log (error_message, timestamp) VALUES (SQLERRM, CURRENT_TIMESTAMP());
END;
```

```
B.
BEGIN
  GRANT USAGE ON DATABASE app_db TO APPLICATION ROLE app_public;
  GRANT USAGE ON SCHEMA app_db.app_schema TO APPLICATION ROLE app_public;
  GRANT EXECUTE MANAGED TASK ON MANAGED TASK app_db.app_schema.daily_data_processing TO APPLICATION ROLE app_public;
EXCEPTION
  WHEN OTHER THEN
    INSERT INTO app_db.app_schema.execution_log (error_message, timestamp) VALUES (SQLERRM, CURRENT_TIMESTAMP());
```

```
C. END;
GRANT USAGE ON DATABASE app_db TO APPLICATION ROLE app_public;
GRANT USAGE ON SCHEMA app_db.app_schema TO APPLICATION ROLE app_public;
```

D. GRANT EXECUTE MANAGED TASK ON MANAGED TASK app_db.app_schema.daily_data_processing TO APPLICATION ROLE app_public;

E. GRANT EXECUTE MANAGED TASK ON MANAGED TASK TO APPLICATION ROLE app_public WITH GRANT OPTION;

Answer: (SHOW ANSWER)

Option C is the most comprehensive. It grants the necessary USAGE privileges on the database and schema, which are prerequisites for the consumer account's application role to access the managed task. It also includes robust error handling using a BEGIN...EXCEPTION block, logging any errors encountered during the privilege granting process to the 'execution_log' table. Options A and D are missing the necessary USAGE privileges, which would prevent the task from executing. Option B lacks USAGE grants and only handles errors within the GRANT execution itself, not potential issues with missing

database/schema access for the role. Option E includes 'WITH GRANT OPTION' , which is unnecessary and potentially insecure as it allows the consumer to further grant the EXECUTE MANAGED TASK privilege.

NEW QUESTION: 2

You are building a Snowflake Native Application that relies on a complex transformation pipeline. This pipeline involves multiple SQL scripts and User-Defined Functions (UDFs). To manage this complexity, you want to ensure that the deployment process is automated and repeatable. Which approach would you consider to manage the deployment and versioning of your application code and its database objects?

- A.** Manually execute each SQL script and UDF creation statement in the consumer's Snowflake environment.
- B.** Package all SQL scripts and UDFs into a single, large SQL file and execute it during installation.
- C.** Use a deployment tool like Flyway or Liquibase to manage database schema changes and versioning, integrating it into your application's setup script.
- D.** Store SQL Scripts and UDFs external to the application, providing instructions to the consumers to manually download and install.
- E.** Rely exclusively on Snowflake's built-in version control features (if available), which automatically manage schema changes.

Answer: ([SHOW ANSWER](#))

Using a deployment tool like Flyway or Liquibase (C) enables you to manage database schema changes and versioning in an automated and repeatable manner. These tools allow you to define migrations, apply them in a controlled sequence, and track changes over time. Executing scripts manually (A) is prone to errors and not scalable. A single large SQL file (B) becomes difficult to manage and troubleshoot. Storing SQL scripts external to the application (D) increases risk as the application is tightly linked to external artifacts. Snowflake itself does not provide build-in schema version control(E).

NEW QUESTION: 3

You are developing a Snowflake Native Application that processes customer data. To comply with data residency requirements, you need to ensure the application processes data within the customer's Snowflake account. The application logic resides in a Snowpark Python stored procedure. During development, you test this procedure using the 'CALL' command. However, after installing the application in a customer's account, the procedure fails with a 'privilege insufficient' error. What are the MOST likely reasons for this and how can you rectify them within the package definition?

- A.** The stored procedure is missing the EXECUTE MANAGED ACCOUNT privilege. Add 'GRANT EXECUTE MANAGED ACCOUNT ON PROCEDURE TO APPLICATION ROLE' to the setup script.
- B.** The application role lacks sufficient privileges on the data sources within the customer's account. Grant the appropriate privileges (e.g., 'SELECT' , 'INSERT') on the required tables/views to the application role in the setup script using 'GRANT SELECT ON TABLE TO APPLICATION ROLE'.
- C.** The procedure owner is not the application. This can be resolved by transferring ownership to the application role using 'ALTER PROCEDURE (...) OWNER TO APPLICATION ROLE' in the setup script.
- D.** The application manifest is missing the 'external_network_access' property set to 'true'. Add this to the manifest and use network rules if the application requires access to external resources.
- E.** The stored procedure requires the 'IMPORTED PRIVILEGES' on the database. Grant this using 'GRANT IMPORTED PRIVILEGES ON DATABASE TO APPLICATION ROLE' in the setup script.

Answer: ([SHOW ANSWER](#))

When a Snowflake Native Application runs in a customer's account, it operates under the security context of an application role defined within the application package. The 'privilege insufficient' error suggests the application role lacks the necessary permissions to execute the stored procedure or access underlying data. Option A correctly identifies the need for the EXECUTE MANAGED ACCOUNT privilege for stored procedures. Option B highlights the importance of granting specific data access privileges to the application role on the customer's data sources. Option C is incorrect because the procedure owner doesn't need to be changed for correct application working. Option D pertains to access to external networks, not internal privileges. Option E is incorrect because imported privileges are not required to execute stored procedure.

NEW QUESTION: 4

A Snowflake Native App is designed to process data in a consumer's account. The application requires access to multiple shared databases provided by different providers. The application provider configures their application. Snowflake is configured so that the app can see all of these shared databases. Now, to enable the Native App with the privileges it requires, what are the necessary steps to allow the application's role, , to query tables in these shared databases?

- A.** Only grant 'IMPORTED PRIVILEGES' on each shared database to the 'app_role' .
- B.** Grant 'IMPORTED PRIVILEGES' on each shared database to the 'app_role' . Additionally, grant 'SELECT' privilege on each specific table or view the application needs to access in each shared database to the 'app_role' .

- C. Grant USAGE on each shared database to the 'app_role' . Additionally, grant 'SELECT privilege on each specific table or view the application needs to access in each shared database to the 'app_role' .
- D. Grant 'ALL PRIVILEGES on each shared database to the 'app_role'. This will allow the application to access all objects within the shared databases.
- E. No additional steps are required. Once the application is installed, it automatically has access to all shared databases.

Answer: (SHOW ANSWER)

The correct answer is B. While 'IMPORTED PRIVILEGES allows the application to see the shared database, it doesn't automatically grant access to the data within. You must also grant the specific 'SELECT privilege on the tables and views that the application requires. A is incomplete since it is needed, but not sufficient. C is incorrect as USAGE does not provide SELECT ability. D violates the least privilege principle, and E is incorrect because privileges must be explicitly granted.

NEW QUESTION: 5

You are developing a Snowflake Native Application that processes financial transactions. You want to implement a robust version management strategy. You have created versions 'v1_0', 'v1_1', and 'v2_0'. Version 'v1_1' has a critical bug fix and needs to be prioritized for immediate deployment to specific consumer accounts that reported issues, while 'v2_0' contains significant feature enhancements and is undergoing a phased rollout. What is the most efficient and secure way to manage these releases in Snowflake?

- ☐ Grant the 'USAGE' privilege on the 'v1_1' version to the specific consumer accounts needing the bug fix and grant 'USAGE' on 'v2_0' to accounts participating in the phased rollout. All other accounts get 'USAGE' on 'v1_0'.
- ☐ Create separate application packages for each version ('v1_0', 'v1_1', 'v2_0') and grant appropriate privileges to consumer accounts based on their assigned version. This involves deploying the application multiple times.
- ☐ Utilize the 'DEFAULT VERSION' property of the application package to control the version exposed to new installations. Consumers on old version need to manually upgrade from Snowflake Marketplace.
- ☐ Create a shared secure view on top of the different versions and use conditional logic within the view to serve different data based on the consumer account. This requires modifying application code for version handling.
- ☐ Use the 'PATCH' command to apply the bug fix from 'v1_1' to 'v1_0' and then upgrade all consumer accounts to the patched 'v1_0'. Initiate a phased rollout of 'v2_0' afterwards.

A. Option A

B. Option B

C. Option C

D. Option D

E. Option E

Answer: (SHOW ANSWER)

Option A is the most efficient. Granting 'USAGE' on specific versions directly controls which version each consumer account accesses. This avoids the overhead of deploying multiple application packages and doesn't require complex data manipulation within the application itself. Option B is inefficient and resource intensive. Option C will only effect the new installation, and we already have consumers that need the bug fixed on the old version. Option D is complex and requires custom code which can lead to errors. Option E is viable but the patch command is suited for more than just bug fixes, using the version grants are the best method for this case.

NEW QUESTION: 6

You are designing the setup script for a Snowflake Native Application. This application requires the creation of several views based on data residing in the consumer's account. These views need to be dynamically created based on a configuration table that is part of your application package. The configuration table contains view names, the underlying tables in the consumer's account, and the specific SQL queries for each view. The design must adhere to best practices for security and data governance. Which approach would you use for designing this setup script, considering security vulnerabilities and execution context?

- A.** Directly construct and execute the 'CREATE VIEW' statements within the setup script using string concatenation, reading the view definitions from the configuration table. This minimizes code complexity and ensures immediate view creation.
- B.** Employ parameterized SQL (using '?' 7 placeholders) within the 'CREATE VIEW' statements, binding the view names and queries read from the configuration table. Execute these statements using 'EXECUTE IMMEDIATE' to prevent SQL injection vulnerabilities.
- C.** Use dynamic SQL generation with proper escaping functions (e.g., to prevent SQL injection. Store the generated SQL in a temporary table, and then execute each statement from the temporary table using 'EXECUTE IMMEDIATE
- D.** Use a stored procedure, owned by the application, to dynamically generate and execute the 'CREATE VIEW' statements. The stored procedure reads the view definitions from the configuration table and creates the views using dynamic SQL.
- E.** Create a series of static 'CREATE VIEW' statements within the setup script, each targeting a predefined table in the consumer's account. The application will handle data filtering within these views.

Answer: ([SHOW ANSWER](#))

Option D represents the most secure and maintainable approach. By using a stored procedure owned by the application, you encapsulate the dynamic SQL generation and execution within a controlled environment. This reduces the risk of SQL injection vulnerabilities because the stored procedure is executed with the privileges of the application, and not the installer or end user. Parameterized SQL (Option B) is not directly supported for object names or complex SQL structures in Snowflake. String concatenation (Option A) is highly vulnerable to SQL injection. Using a temporary table (Option C) adds unnecessary complexity without significantly improving security. Static views (Option E) are inflexible and don't address the requirement of dynamic view creation.

NEW QUESTION: 7

You are designing a Snowflake Native App that processes sensitive customer data. To comply with data residency requirements, you need to ensure the application processes data exclusively within the consumer's Snowflake account and does not transmit data outside.

Which features of the Snowflake Native App Framework support this requirement? (Select all that apply)

- A.** Secure Data Sharing: Allows the application to access data directly within the consumer's account without copying or moving it.
- B.** External Functions: Enables the application to call external services for data processing and enrichment, ensuring all data remains within the Snowflake environment.
- C.** Snowflake Marketplace: Provides a secure and governed platform for distributing the application without requiring direct access to the consumer's account.
- D.** Stored Procedures: Allows the application to execute custom logic within the consumer's Snowflake environment, processing data locally.
- E.** Application Packages: Package the entire application including code and configuration, ensuring data locality and not data transfer outside environment

Answer: ([SHOW ANSWER](#))

The correct answers are A, D and E. Secure Data Sharing enables direct access to data without movement. Stored Procedures execute logic within the consumer's environment. Application packages ensure everything within the consumer's environment. External Functions (B) inherently involve sending data outside of Snowflake, violating the residency requirement. Snowflake Marketplace (C) is about distribution and governance, not data processing location.

NEW QUESTION: 8

After publishing your Snowflake Native Application on the Marketplace, you discover that a critical security vulnerability has been identified in a third-party library included in your application package. You need to quickly address this issue and push an updated version to your consumers. What is the recommended sequence of steps to remediate this vulnerability with minimal disruption to your users?

- A.** 1. Deprecate the current version of the application. 2. Develop and test a new version with the updated library. 3. Publish the new version to the Marketplace. 4. Announce the new version to consumers.
- B.** 1. Create a patch for the existing application. 2. Deploy the patch to all consumer accounts. 3. Notify consumers of the updated patch.
- C.** 1. Develop and test a new version with the updated library. 2. Publish the new version as a private listing to selected consumers for testing. 3. Once validated, publish the new version to the Marketplace. 4. Deprecate the vulnerable version. 5. Announce the new version to consumers.
- D.** 1. Immediately unlist the current version of the application. 2. Develop and test a new version with the updated library. 3. Publish the new version to the Marketplace. 4. Announce the new version to consumers.
- E.** 1. Develop and test a new version with the updated library. 2. Publish the new version to the Marketplace. 3. Deprecate the vulnerable version. 4. Announce the new version to consumers.

Answer: ([SHOW ANSWER](#))

The recommended approach prioritizes a swift update with minimal disruption. Developing and testing a new version, publishing it, deprecating the old version, and then announcing it ensures continuous availability. Unlisting (D) or immediately deprecating (A) the application causes disruption. Patches (B) are not the standard procedure for Native Apps. A private listing (C) adds unnecessary complexity in this urgent scenario.

NEW QUESTION: 9

A Native App provider has created an application package and wants to control the versions of the application available to different consumers. They need to ensure that a specific consumer account, 'CUSTOMER R, only has access to version '1.0' of the application. Which of the following steps are necessary to achieve this?

- A. No specific steps are required. By default, consumers will automatically receive the latest version of the application.
- B. Specify the version '1.0' when creating the share object for 'CUSTOMER_A'.
- C. When creating the authorization grant for the APPLICATION ROLE, specify VERSION = '1.01'.
- D. Create a new application package specifically for 'CUSTOMER A' and only include version '1.0' in that package.
- E. The provider must first create a new version of the application package containing only version 1 .C , and then create an authorization grant for the role to CUSTOMER A that includes version '1.0'

Answer: (SHOW ANSWER)

The correct approach is to specify the VERSION parameter within the CREATE AUTHORIZATION GRANT statement. This allows the provider to control which version of the application a specific consumer is authorized to use. Creating separate packages or shares is not the intended mechanism for version control within the Native App Framework. It's important to grant authorization to application roles.

NEW QUESTION: 10

A Native Application provider has created a database named 'app_db' using 'CREATE DATABASE app_db WITH MANAGED ACCESS;'. They now need to create a secure view that exposes a subset of data from an underlying table 'sensitive_data' within the schema 'internal_schema' of 'app_db' to a consumer's role named 'consumer_role'. They wish to minimize data exposure while adhering to managed access principles. Which set of SQL statements represents the MOST secure and correct approach?

- A. CREATE VIEW app_db.internal_schema.consumer_view AS SELECT coll , c012 FROM app_db.internal_schema.sensitive_data; GRANT SELECT ON VIEW app_db.internal_schema.consumer_view TO ROLE consumer_role;
- B. CREATE OR REPLACE SECURE VIEW app_db.internal_schema.consumer_view AS SELECT coll , c012 FROM app_db.internal_schema.sensitive_data; GRANT SELECT ON VIEW TO ROLE consumer_role;
- C. CREATE OR REPLACE VIEW app_db.internal_schema.consumer_view AS SELECT coll , c012 FROM app_db.internal_schema.sensitive_data; GRANT ALL PRIVILEGES ON VIEW TO ROLE consumer_role;
- D. CREATE OR REPLACE SECURE VIEW app_db.internal_schema.consumer_view AS SELECT coll , c012 FROM app_db.internal_schema.sensitive_data; GRANT ALL PRIVILEGES ON VIEW TO ROLE consumer_role;
- E. CREATE OR REPLACE SECURE VIEW app_db.internal_schema.consumer_view AS SELECT coll , c012 FROM app_db.internal_schema.sensitive_data; GRANT SELECT ON VIEW TO ROLE consumer_role; REVOKE ALL PRIVILEGES ON VIEW app_db.internal_schema.consumer_view FROM ROLE PUBLIC;

Answer: (SHOW ANSWER)

Creating a 'SECURE VIEW is essential. A secure view hides the underlying data and implementation details from the consumer. The 'OR REPLACE clause ensures idempotency. Granting only 'SELECT is the principle of least privilege. Secure views ensure data security and integrity while working within the constraints of MANAGED ACCESS. Option A, missing the SECURE keyword, doesn't hide the underlying data. C and D have grant all, which goes against least privilege. E, while attempting to add more security with REVOKE ALL, is unnecessary given the 'WITH MANAGED ACCESS property. B is the most secure and appropriate solution.

NEW QUESTION: 11

You are managing versions of a Snowflake Native App that persists data to tables within the consumer's account. You want to provide consumers with the ability to rollback to a previous version of the application without losing data. Which combination of Snowflake features and practices would best support this requirement?

- A.** Rely solely on Snowflake's Time Travel feature on the consumer's tables, instructing users to manually restore data to the desired point in time after rolling back the application.
- B.** Implement a backup and restore mechanism within your application, creating regular snapshots of the consumer's tables and storing them in a secured location. Provide a rollback procedure that restores data from the appropriate snapshot.
- C.** Store data version history within the application and reconstruct historical states on demand. This requires complex data modeling and coding.
- D.** Combine Snowflake's Time Travel feature with automated scripts within your application that identify and restore tables affected by the rollback to the specific point in time when the previous version was active. Provide users with a simplified rollback process.
- E.** Require consumers to manually back up their data before each application upgrade and provide instructions on how to restore it.

Answer: ([SHOW ANSWER](#))

Combining Time Travel with automated scripts (D) gives consumers a simple process and maintains data integrity. The backup and restore mechanism (B) adds an extra layer of protection, and could potentially allow for longer retention than the standard Time Travel configuration. (A) places too much burden on the consumer. (C) is too complex. (E) is not user friendly and prone to errors. Implementing scripts to manage time travel on the consumers' data warehouse is more effective than relying on them to do it themselves.

NEW QUESTION: 12

You are developing a Snowflake Native App that leverages shared data from the consumer's account. The app needs to dynamically determine the existence of a specific table (e.g., 'CUSTOMER DATA) in the consumer's shared database before attempting to query it. Which of the following SQL commands or approaches within the application package provides the MOST reliable way to programmatically check for the existence of the table?

- A.** Use the ' SHOW TABLES LIKE 'CUSTOMER_DATA" command and check if any rows are returned.
- B.** Execute a 'SELECT 1 FROM CUSTOMER_DATA LIMIT 1' query within a 'TRY...CATCH' block and handle the exception if the table does not exist.
- C.** Use the 'INFORMATION SCHEMA.TABLES' view and query it to check if a table with the name 'CUSTOMER DATA' exists in the shared database.
- D.** Use the 'DESCRIBE TABLE CUSTOMER_DATX command within a 'TRY...CATCH' block and handle the exception if the table does not exist.
- E.** Attempt to grant SELECT privilege on the table and check if an error occurs

Answer: ([SHOW ANSWER](#))

Option C (Use the 'INFORMATION SCHEMATA.BLOCKS' view) is the most reliable and recommended approach. The INFORMATION_SCHEMA provides metadata about database objects, and querying it to check for the existence of a table is a standard and efficient practice. Option A ('SHOW TABLES') might be affected by the session context or user's privileges. Options B and D (using 'TRY...CATCH' blocks) rely on exception handling, which can be less efficient and less readable. Also, consuming exceptions can be more expensive than directly querying system views. Option E is inefficient since it requires an attempt to change a permission.

NEW QUESTION: 13

You've installed a Snowflake Native Application that provides advanced data analytics. The application requires a consumer-defined function (UDF), 'MASK SENSITIVE DATA, to be created in their account and used by the application. The application's setup script includes logic to check for the UDF's existence, but you want to ensure the application gracefully handles the case where the consumer has not yet created the UDF. Which of the following strategies can you implement to improve the application's resilience and provide a better user experience in this scenario? Select all that apply.

- A.** In the application's setup script, create a placeholder UDF with the same name and signature as 'MASK SENSITIVE DATA' that returns 'NULL'. The application logic can then handle 'NULL' values appropriately until the consumer provides the actual UDF.
- B.** Within the application's SQL code, use the 'TRY FUNCTION' function to call 'MASK SENSITIVE DATA'. If the UDF does not exist, 'TRY FUNCTION' will return 'NULL' instead of raising an error.
- C.** Implement error handling within the application's code to catch the 'SQL compilation error: UDF does not exists exception. Provide a user-friendly error message and instructions on how to create the UDF.
- D.** Require the consumer to create the UDF before installing the application. The application setup script should fail if the UDF is not found, preventing installation.
- E.** Dynamically generate the application code using a stored procedure, only including the parts of the code that rely on if the function exists. Use 'SYSTEM\$GET PRIVILEGES()' to check existence.

Answer: ([SHOW ANSWER](#))

Options B and C are the best strategies for handling the missing UDF gracefully. 'TRY_FUNCTION' allows the application to continue running even if the UDF is not present, while error handling provides a clear and informative message to the user. Option A is problematic because it introduces a UDF that is not the intended function, potentially leading to data inconsistencies. Option D creates a rigid dependency and a poor user experience. Option E adds unnecessary complexity.

NEW QUESTION: 14

You are preparing to publish a Snowflake Native Application on the Snowflake Marketplace. You have successfully built and tested the application in your development account. Which of the following steps are CRITICAL for preparing the application listing metadata and ensuring a smooth onboarding experience for consumers?

- A. Provide a comprehensive and user-friendly application description that clearly outlines the application's functionality, benefits, and intended use cases. Include screenshots and/or videos demonstrating the application in action.
- B. Define a clear pricing model (e.g., free, per-use, subscription) and accurately estimate the resource consumption (Snowflake credits) required for typical application usage. Include a free trial period if possible.
- C. Thoroughly test the application installation and upgrade process in a separate Snowflake account (ideally one that mimics a typical consumer environment) to identify and resolve any potential issues.
- D. Ensure all necessary data governance and security measures are in place, including compliance with relevant regulations (e.g., GDPR, CCPA). Clearly document these measures in the application listing.
- E. The listing should include a well-defined support strategy, outlining how you will handle customer inquiries, bug reports, and feature requests. Provide contact information and estimated response times.

Answer: (SHOW ANSWER)

All of the listed steps are crucial for preparing a Snowflake Native Application for the Marketplace. A detailed description (A) attracts users and sets expectations. A clear pricing model (B) is essential for monetization. Thorough testing (C) ensures a smooth user experience. Data governance and security (D) build trust and are often legally required. A support strategy (E) demonstrates commitment and helps retain users. Neglecting any of these aspects can negatively impact the application's success on the Marketplace.

NEW QUESTION: 15

You are developing a Snowflake Native Application that performs complex data transformations. As part of your development lifecycle, you need to ensure that your code behaves idempotently even when facing intermittent network errors or unexpected disruptions. Consider the following function written in Snowpark Python that updates a table:

```
import snowflake.snowpark.functions as F
from snowflake.snowpark import Session
```

```
def update_table(session: Session, table_name: str, df: snowflake.snowpark.DataFrame):
    df.write.mode('overwrite').save_as_table(table_name)
```

What modification is necessary to make this function idempotent against disruptions during the writing process to Snowflake and unexpected execution?

- A. Use `df.write.mode('append').save_as_table(table_name)` instead of `'overwrite'`.
- B. Wrap the `df.write` operation in a retry mechanism with exponential backoff to handle intermittent network errors.
- C. Create a temporary table, write the transformed data to the temporary table, and then atomically rename the temporary table to the original table name.
- D. Use a `'MERGE'` statement to selectively update only changed rows, while ignoring existing ones.
- E. Leverage Snowflake's `'TRANSACTIONAL_REPLICATION'` to replicate the operations to minimize data loss in between failures.

Answer: (SHOW ANSWER)

Option C provides the best approach to guarantee idempotency. Creating a temporary table, writing the data to it, and then atomically renaming it ensures that the table is either fully updated or remains in its original state. If failures happen after the creation of temporary table and before it is renamed (a failure which is assumed to be very rare), then, after code recovery and execution, the table would be first created and then atomically renamed, thus achieving idempotency. Retrying writes (B) can help with network errors but doesn't inherently guarantee idempotency if the write partially succeeds before the error. 'append' (A) leads to duplicate data in the event of re-execution. 'MERGE' (D) might lead to data inconsistency depending on the transformation logic. 'TRANSACTIONAL REPLICATION' (E) is a good approach for disaster recovery but will not help to resume the execution, after recovering the code after failure, at the point where it failed before.

NEW QUESTION: 16

You're developing a Snowflake Native App with a Streamlit I-JL. The app needs to read configuration data from a file named within your application package. The file contains sensitive information, and you want to ensure it's only accessible by your application's code and not directly readable by the consumer. What steps would you take to accomplish this?

- A. Store 'config.json' in an internal stage and grant the 'READ' privilege on the stage to the consumer's role.
- B. Store 'config.json' within the application package and read it using Python code within a stored procedure. Grant appropriate privileges on the stored procedure, but not on the 'config.json' file or the internal stage.
- C. Store 'config.json' in a table within your application's schema and grant the 'SELECT' privilege on the table to the consumer's role.
- D. Store 'configjson' outside the application package and read it directly from an external stage at runtime.
- E. Store 'config.json' within the Streamlit code, and then encode and decode it to the original string when the user invokes the Streamlit application.

Answer: (SHOW ANSWER)

The best approach is to store the 'config.json' file within the application package and access it using Python code within a stored procedure. By granting privileges only on the stored procedure and not on the 'config.json' file itself or the internal stage, you ensure that the consumer cannot directly read the configuration data. Option A exposes it directly to the consumer. Option C would need to be encrypted, which adds complexity. Option D means the configuration is outside your application package. Option E is simply obfuscation and not encryption and hence should be avoided.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 17

You are deploying a Snowflake Native Application that utilizes a UDF (User-Defined Function) to perform complex data transformations. The UDF is written in Python and depends on several external packages. You need to ensure that the consumer of your application can seamlessly install and use the UDF without encountering dependency conflicts or versioning issues. How should you manage the UDF and its dependencies during the deployment process, considering Snowflake's Native App framework?

- A. Package the Python UDF and its dependencies into a ZIP file and include it in the application package. Consumers will need to manually install the dependencies using 'conda install' after installing the application.
- B. Use Anaconda packages within the Snowflake environment to manage Python dependencies. Ensure all required packages are explicitly listed in the setup script used during the application installation process.
- C. Upload the Python UDF code directly into a stage and create a Snowflake function using the 'CREATE FUNCTION' command. Rely on the consumer's existing Python environment to resolve dependencies.
- D. Utilize the Snowflake Native Apps framework to bundle the Python UDF and its dependencies within the application package. Snowflake will automatically handle dependency resolution during application installation.
- E. Document the Python UDF dependencies in the application's documentation and instruct consumers to manually install them before using the function. No packaging is required.

Answer: (SHOW ANSWER)

The Snowflake Native Apps framework is designed to handle dependency management automatically. Option D is the correct answer. The framework ensures that the UDF and its dependencies are packaged together and resolved seamlessly during installation, avoiding conflicts and versioning issues for the consumer. Options A, C, and E place the burden of dependency management on the consumer, which is not ideal. While B uses Anaconda packages, it's less seamless and automatic than using the Native Apps framework's built-in capabilities. This automatic dependency resolution is a key benefit of the Native Apps framework.

NEW QUESTION: 18

A company publishes a Snowflake Native Application on the Snowflake Marketplace providing aggregated sales data. They want to ensure data is fresh and updated daily at 00:00 UTC. They are considering several approaches to automate this data refresh process.

Which of the following methods are MOST effective and SECURE for scheduling and executing this daily data update within the Snowflake Native Application environment? Choose two.

- A. Schedule a Python script using an external task service (e.g., AWS Lambda) that connects to Snowflake and executes the data refresh procedure.
- B. Use Snowflake Tasks scheduled to run daily at 00:00 UTC to execute a stored procedure within the Native Application that updates the data.

- C. Utilize a cron job on a virtual machine to execute a SQL script via SnowSQL, connecting to Snowflake and refreshing the data.
- D. Leverage Snowflake's Data Marketplace update scheduling feature (if available and applicable to their type of listing) to automatically refresh the data.
- E. Implement a Java-based scheduler within the Snowflake Native Application itself using UDFs, triggered by an external service via API integration.

Answer: (SHOW ANSWER)

Snowflake Tasks provide a secure and managed way to schedule tasks within the Snowflake environment. Leveraging Data Marketplace update scheduling (if available) is the most direct and efficient method. External task services and cron jobs introduce external dependencies and security concerns, and Java schedulers within UDFs are less common and can be complex to manage.

NEW QUESTION: 19

You are building a Snowflake Native App that aggregates data across multiple consumer accounts. Your app needs to securely access data in each consumer's Snowflake account without requiring them to share their credentials directly. You are evaluating different options for secure data access. Which of the following strategies are viable and recommended to implement this, maximizing security and minimizing administrative overhead for the consumer?

- A. Require consumers to grant SELECT privileges on their tables directly to the application's account. This necessitates the consumer to manage grants directly.

```
GRANT SELECT ON TABLE consumer_db.schema.table TO APPLICATION ;
```

- B. Use Snowflake Data Clean Rooms to allow the app to query data from the consumers accounts.

```
CREATE OR REPLACE DATA CLEAN ROOM consumer_db.public.clean_room;
```

```
GRANT USAGE ON DATABASE consumer_db TO SHARE WITH APPLICATION ;
```

- C. Implement a data sharing agreement where consumers create a share from their account and grant usage to your application's account using secure data sharing. The application account can then create a database from this share.

```
-- Consumer account:
```

```
CREATE SHARE consumer_db_share;
```

```
GRANT USAGE ON DATABASE consumer_db TO SHARE consumer_db_share;
```

```
GRANT SELECT ON TABLE consumer_db.schema.table TO SHARE consumer_db_share
```

```
ALTER SHARE consumer_db_share ADD ACCOUNTS = ;
```

```
-- Application provider account:
```

```
CREATE DATABASE app_db FROM SHARE .consumer_db_share;
```

- D. Have consumers create and share a JWT (JSON Web Token) with your application. The JWT will be used to authenticate against their Snowflake instance via an external function.

```
-- Consumer account:
```

```
CREATE OR REPLACE FUNCTION generate_jwt() RETURNS STRING ...;
```

```
SELECT generate_jwt();
```

```
-- Application account:
```

```
CREATE OR REPLACE EXTERNAL FUNCTION validate_jwt(jwt STRING) RETURNS BOOLEAN ...;
```

- E. Utilize Snowflake's Native App data sharing capabilities where the provider app shares data products back to the consumer account and the consumer account can use this inside their apps.

```
-- Provider account (within setup script):
```

```
CREATE OR REPLACE DATA PRODUCT dp_example;
```

```
GRANT OWNERSHIP ON DATA PRODUCT dp_example TO APPLICATION ROLE app_role;
```

```
-- Consumer account:
```

```
SHOW DATA PRODUCTS;
```

Answer: C,E (LEAVE A REPLY)

Granting SELECT directly (A) requires the consumer to manage grants directly, which is not ideal for scalability or security. Passing JWT tokens (D) is complex and requires careful management of keys. Snowflake Data Clean Rooms(B) are not the correct solution to get consumer data in Provider App.. Secure data sharing (C) is a viable and secure option, as it allows consumers to share data without exposing credentials. Snowflake's Native App sharing Data Products back to the consumer account (E) is the best option for secure data access.

NEW QUESTION: 20

A Snowflake Native App installed in a consumer account needs to access data in a specific table within the consumer's account to perform its core functionality. Which of the following is the MOST secure and recommended way for the application to gain access to this data? (Assume the application developer does not have direct access to the consumer's Snowflake account.)

- A. The consumer grants the 'GLOBAL' privilege 'SELECT on the table directly to the application package name.
- B. The application provider requests the 'IMPORTED PRIVILEGES on the consumer account, allowing the application to access any object within the account.
- C. The consumer creates a secure view on top of the table and grants 'SELECT privilege on the secure view to the application's installed application object (application instance).
- D. The consumer shares the table directly with the application provider's Snowflake account.
- E. The application prompts the consumer for their Snowflake account credentials and uses those credentials to access the data.

Answer: [\(SHOW ANSWER\)](#)

The MOST secure approach is to create a secure view on top of the table and grant 'SELECT privilege on the secure view to the installed application. This limits the application's access to only the data exposed by the view, minimizing the potential security risk. Granting access directly to the application package (option A) is not supported. Requesting 'IMPORTED PRIVILEGES (option B) is too broad and insecure. Sharing the table with the provider (option D) is not necessary and adds complexity. Asking for credentials (option E) is a major security violation. Secure Views prevent direct access to underlying tables while still allowing applications to function as intended.

NEW QUESTION: 21

An organization is building a Snowflake Native App that performs complex data transformations. The transformation logic is encapsulated in a stored procedure. They want to allow consumer accounts to execute this stored procedure, but ONLY with the privileges of the application itself, preventing access to the consumer's data through the procedure. Which set of actions are required to achieve this?

- A. Create the stored procedure with the 'EXECUTE AS OWNER clause and grant the 'USAGE privilege on the application package to the consumer's role.
- B. Create the stored procedure with the 'EXECUTE AS CALLER clause and grant the 'APPLY APPLICATION' privilege on the application package to the consumer's role.
- C. Create the stored procedure with the ' EXECUTE AS OWNER clause and grant the ' EXECUTE APPLICATION' privilege on the application instance to the consumer's role.
- D. Create the stored procedure with the ' EXECUTE AS CALLER clause and grant the 'EXECUTE APPLICATION' privilege on the application instance to the consumer's role.
- E. Create the stored procedure with the ' EXECUTE AS OWNER' clause and grant the ' IMPORTED PRIVILEGES' on the application package to the consumer's role.

Answer: [C \(LEAVE A REPLY\)](#)

The correct answer is C. Using 'EXECUTE AS OWNER ensures the stored procedure runs with the privileges of the application owner, isolating it from the consumer's privileges. Granting ' EXECUTE APPLICATION' allows the consumer's role to execute the application and its associated procedures within the security context of the application . 'EXECUTE AS CALLER would cause the procedure to run with the privileges of the consumer, defeating the isolation goal. 'USAGE, 'APPLY APPLICATION' and 'IMPORTED PRIVILEGES do not allow executing application.

NEW QUESTION: 22

A Native Application deployed on Snowflake uses a warehouse 'APP WH'. You want to monitor the warehouse usage costs specifically attributed to this application. Which of the following approaches provides the MOST granular and accurate way to track these costs?

- A. Examine the 'WAREHOUSE METERING HISTORY view in the 'SNOWFLAKE database without any filtering.
- B. Tag all queries originating from the application with a specific tag and then filter the 'QUERY_HISTORY view based on this tag and the warehouse name.
- C. Create a separate warehouse solely for the application and then examine the 'WAREHOUSE METERING HISTORY view.
- D. Use the view and filter by 'USER_NAME associated with the application role.
- E. Rely on the total credits consumed by the account, as the application's usage is insignificant.

Answer: [C \(LEAVE A REPLY\)](#)

Creating a separate warehouse exclusively for the application is the most precise method. The view then provides accurate usage data. Tagging queries (Option B) is a viable alternative but requires diligent implementation and maintenance of the tagging strategy. Filtering 'QUERY_HISTORY' by (Option D) may not accurately reflect costs if the role is used for other purposes. Examining without filtering (Option A) is too broad, and assuming the application's usage is insignificant (Option E) is incorrect for proper cost management.

NEW QUESTION: 23

A Snowflake Native App developer is designing the upgrade process for their application. They need to ensure that data stored in the application's managed tables is preserved during the upgrade and that the application remains functional throughout the upgrade process. Which of the following strategies should the developer implement to achieve these goals? (Select TWO)

- A. Use the 'DROP TABLE' command with the 'CASCADE' option to remove all tables and views before deploying the new version of the application.
- B. Implement a versioning system for the application's metadata using the 'APPLICATION VERSION' construct, ensuring that the new version can handle data from previous versions.
- C. Utilize Snowflake's data sharing capabilities to create a read-only replica of the application's data for the new version to access during the upgrade.
- D. Design the application's data model to be backward-compatible, allowing the new version to read and write data in the format used by previous versions. Use 'ALTER TABLE' to add new columns or modify existing ones without breaking existing functionality.
- E. Before upgrading, unload all data from the application's tables to cloud storage, and then reload the data into the new version after the upgrade is complete.

Answer: ([SHOW ANSWER](#))

Option B correctly describes the use of 'APPLICATION VERSION' to manage metadata compatibility between versions. Option D correctly emphasizes designing for backward compatibility in the data model using 'ALTER TABLE' for schema changes, which avoids data loss and ensures a smoother upgrade process.

NEW QUESTION: 24

You are developing a Snowflake Native Application that uses Streamlit to provide a user interface for data analysts. The application accesses sensitive customer data residing in a managed database. To enhance security, you want to implement row-level security (RLS) based on the user's role. You have the following Streamlit code snippet:

```
import streamlit as st
import snowflake.connector

@st.cache_data(ttl=600)
def get_data(query):
    ctx = snowflake.connector.connect(st.secrets["snowflake"])
    cur = ctx.cursor()
    cur.execute(query)
    data = cur.fetchall()
    cur.close()
    ctx.close()
    return data

query = "SELECT * FROM customer_data"
data = get_data(query)
st.dataframe(data)
```

Which of the following actions should you take to implement RLS in the Snowflake Native App, ensuring only authorized users can view specific rows of 'customer_data'? Assume the already contains necessary credentials. Choose the most secure and efficient method.

- A. Modify the SQL query within the 'get_data' function to include a 'WHERE' clause that filters data based on the current Streamlit user's username obtained using 'st.session_state.user.username'.
- B. Create a secure view in Snowflake that implements RLS based on the role of the user executing the query. Grant the Streamlit user access to this view instead of the base table, and modify the 'get_data' function to query the view.

- C.** Implement RLS directly within the Streamlit application by filtering the 'data' DataFrame based on the user's role before displaying it using 'st.dataframe'. Store user roles in a separate table and fetch them using a different database connection.
- D.** Use Snowflake's masking policies combined with a secure view. The masking policy reveals data based on the user's role. Grant the Streamlit application's role access to the secure view.
- E.** Implement a custom Python decorator that checks user roles before executing the 'get_data' function. Raise an exception if the user lacks the necessary role.

Answer: ([SHOW ANSWER](#))

Using Snowflake's masking policies and secure views is the most secure and efficient method. It centralizes RLS within Snowflake, leveraging its security features. Option A is vulnerable to SQL injection if 'st.session_state.user.username' is not properly sanitized. Option B is better, but masking policies provide more granular control and data obfuscation. Option C pushes security logic to the application layer, which is less secure. Option E adds unnecessary complexity and overhead.

NEW QUESTION: 25

You are developing a Snowflake Native Application intended for the Snowflake Marketplace. The application provides data transformation services, leveraging external functions (UDFs) that invoke a REST API hosted on AWS Lambda. You need to ensure the secure and compliant invocation of this external API from your application while adhering to Snowflake's best practices. Which of the following steps represent the MOST secure and recommended approach for configuring and invoking the external function?

- A.** Create an API integration object in Snowflake that stores the AWS Lambda function's URL and credentials (API key) directly within the integration. Grant the necessary permissions to the application's roles to use the API integration. Define the external function to call the Lambda function URL directly, referencing the API key from the integration object.
- B.** Expose the AWS Lambda function publicly with no authentication. Configure the external function in Snowflake to point to the public Lambda URL. Document to consumers that the API has no security.
- C.** Create an API integration object in Snowflake, configuring a secure authentication mechanism such as AWS IAM roles or OAuth 2.0. Define the external function to use the API integration, providing the necessary parameters for authentication. Configure network policies and secrets (using Snowflake secrets) to secure the connection. Ensure your AWS Lambda function validates the incoming request's identity.
- D.** Store the AWS Lambda function's URL and credentials (API key) in a configuration table within the Snowflake Native Application package. The external function retrieves these credentials from the configuration table at runtime. Configure an alert if the configuration table is modified.
- E.** Use Snowflake's built-in network rules to allow outbound traffic to all AWS IP ranges. Store the API key directly in the UDF definition as an environment variable.

Answer: ([SHOW ANSWER](#))

The MOST secure and recommended approach (Option C) involves creating an API integration object in Snowflake with secure authentication (IAM roles or OAuth 2.0), defining the external function to use the integration, and leveraging network policies and secrets for enhanced security. This aligns with Snowflake's security best practices. Option A is less secure as it stores the API key directly in the integration. Option B is completely insecure. Option D poses a security risk storing credentials within a table. Option E involves blanket outbound access and storing the API key directly in the UDF (bad).

NEW QUESTION: 26

You are developing a Snowflake Native Application package that provides a set of UDFs for data transformation. The UDFs require specific Snowflake privileges to access data in consumer accounts. Which of the following methods can ensure your application has the necessary privileges when installed in a consumer account? (Choose all that apply)

- A.** Request the required privileges in the application package manifest file ('manifest.yml')
- B.** Use 'GRANT OWNERSHIP' to grant the APPLICATION role ownership of the target objects in the consumer account.
- C.** Create a separate role in the consumer account and grant the necessary privileges to that role. Have the application assume this role.
- D.** Leverage 'GRANT APPLICATION ROLE ... ON ...' statements in the setup script to grant the APPLICATION role the required privileges.
- E.** Instruct consumers to manually grant the necessary privileges to the APPLICATION role after installation.

Answer: ([SHOW ANSWER](#))

The application package 'manifest.yml' file is where you declare the resources your application needs and the privileges it requires. The 'GRANT APPLICATION ROLE ON' statement, executed during the setup script, is the correct way to grant the APPLICATION role the necessary privileges on objects in the consumer account. Requiring consumers to manually grant privileges is not a scalable or secure solution. 'GRANT OWNERSHIP' is generally not necessary or recommended as it gives the application full control over the object. Creating a separate role in the consumer account and having the application assume it is an overly complex solution when 'GRANT APPLICATION ROLE' is available.

NEW QUESTION: 27

You are developing a Snowflake Native Application that provides data enrichment services. A consumer reports that a specific query against a view, 'ENRICHED DATA, occasionally returns incorrect results. You suspect a race condition in the underlying stored procedure that populates a temporary table used by the view. Which of the following strategies would be MOST effective in diagnosing and resolving this issue within the application ?

- A. Implement a retry mechanism with exponential backoff in the stored procedure to handle potential concurrency issues when writing to the temporary table.
- B. Replace the temporary table with a permanent table with appropriate locking mechanisms to prevent concurrent modifications. Consumers will need to be notified of a minor version update.
- C. Use Snowflake's 'GET_DDL' function to retrieve the SQL definition of the consumer's query and analyze it for potential errors.
- D. Implement logging within the stored procedure to capture the execution flow and intermediate data values. Use 'SYSTEM\$GET PREDECESSORS to track dependencies. Analyze the logs to identify the race condition. Use a session table rather than a temp table.
- E. Isolate the portion of the stored procedure that writes to the temporary table into a separate stored procedure with 'EXECUTE AS CALLER privileges.

Answer: ([SHOW ANSWER](#))

Options A and D are correct. A retry mechanism can mitigate concurrency issues. Logging helps diagnose the race condition. Option B introduces a permanent table, which changes the app's architecture and might have unintended side effects. Option C examines the consumer's query, but the issue is within the provider's application. Option E is unlikely to resolve the race condition and can introduce security concerns. A session table ensures isolation and mitigates concurrency problems.

NEW QUESTION: 28

A Snowflake Native Application uses a task 'aggregate_data' to aggregate data. This task is owned by the 'app_public' role and executes a stored procedure 'aggregate_data_proc'. Consumers have reported the task sometimes fails intermittently, citing permission issues, specifically related to accessing a specific table in the consumer account. The application provider has granted 'SELECT' privilege on the table to the 'app_public' role. However, the failures persist. What is the MOST LIKELY cause of the intermittent failures related to executing the task?

- A. The stored procedure is executed with definer's rights (OWNER), so the privilege check occurs against the application provider's account, not the consumer's account.
- B. The stored procedure is executed with caller's rights (CALLER), and the 'EXECUTE TASK' privilege wasn't granted correctly, which causes the user to randomly lose access on the Snowflake UI.
- C. The stored procedure is executed with caller's rights (CALLER), but the application role 'app_public' doesn't have sufficient warehouse privileges in the consumer account.
- D. The stored procedure is executed with caller's rights (CALLER), but the application role 'app_public' doesn't have the ACTIVATION privilege on the task.
- E. The task 'aggregate_data' is executing with definer's rights (OWNER) of the application, so the grant of SELECT privilege to 'app_public' is sufficient, and there is likely an issue with transient network connectivity.

Answer: ([SHOW ANSWER](#))

NEW QUESTION: 29

You are developing a Snowflake Native Application that needs to be deployed across multiple Snowflake accounts. The application includes a setup script that creates several tables and stored procedures. However, some consumers might have naming conflicts with the objects created by your setup script. Which of the following strategies can you use to minimize naming conflicts and ensure smooth application deployment? (Select all that apply)

- A. Prefix all objects created by the setup script with a unique, application-specific identifier.
- B. Instruct consumers to drop any conflicting objects before installing the application.
- C. Dynamically generate object names within the setup script using the application's name and a random suffix.
- D. Utilize the 'CREATE OR REPLACE' syntax for all objects in the setup script to overwrite any existing objects.
- E. Package all application objects into a dedicated schema within the consumer's database.

Answer: ([SHOW ANSWER](#))

Prefixing objects with an application-specific identifier provides a clear and consistent way to avoid naming collisions. Dynamically generating object names makes it very unlikely for conflicts to occur.

Packaging objects into a dedicated schema isolates the application's objects and prevents them from interfering with the consumer's existing objects. Asking consumers to drop objects is not user-friendly and can lead to data loss. Using 'CREATE OR REPLACE' might unintentionally overwrite important objects belonging to the consumer.

NEW QUESTION: 30

A Snowflake Native Application developer is debugging a privilege issue in the consumer's account. They suspect that a specific application role, 'app_admin', does not have the necessary privileges to execute a stored procedure 'calculate_summary' within the application. Which of the following SQL statements will BEST help the developer diagnose the problem by listing the privileges granted to the 'app_admin' role? (Assume exists within the application.)

- A. SHOW GRANTS TO ROLE app_admin;
- B. SHOW GRANTS OF APPLICATION ROLE app_admin;
- C. SHOW GRANTS ON STORED PROCEDURE calculate_summary TO APPLICATION ROLE app_admin;
- D. SHOW GRANTS TO APPLICATION app_admin;
- E. SELECT FROM WHERE grantee_name = 'APP_ADMIN';

Answer: (SHOW ANSWER)

'SHOW GRANTS OF APPLICATION ROLE' is the specific syntax used to view the privileges granted to an application role within a Snowflake Native Application. "SHOW GRANTS TO ROLE app_admin" would look for a regular role named app_admin, not an application role. The SNOWFLAKE.ACCOUNT_USAGE views don't know about application roles and can only return data about 'regular' Snowflake roles.

NEW QUESTION: 31

You are developing a Snowflake Native Application using Snowpark Container Services. The application needs to access sensitive data stored in an internal stage, but you want to ensure that only the application's container service can access this stage and no other users or processes. How can you achieve this using network policies and ingress/egress rules?

- A. Create a network policy that allows inbound traffic from all IP addresses and attach it to the account. Grant 'READ and 'WRITE' privileges on the internal stage to the Native App's service account.
- B. Create a network policy that allows inbound traffic only from the specific IP address range of the Snowflake region where the container service is running. Grant READ and 'WRITE' privileges on the internal stage to the Native App's service account.
- C. Create an egress rule for the container service that allows outbound traffic only to Snowflake's internal IP address range. Create a network policy that allows inbound traffic only from the container service's egress IP address. Grant 'READ and 'WRITE' privileges on the internal stage to the Native App's service account.
- D. Create an ingress rule for the container service that allows only outbound traffic to the internet. Grant 'READ and 'WRITE' privileges on the internal stage to the Native App's service account.
- E. Create an ingress rule for the container service that specifies allowed inbound connections and egress rule for the container service that only allows outbound traffic to Snowflake's internal endpoint for stage access. Grant 'READ and 'WRITE' privileges on the internal stage to the Native App's service account.

Answer: (SHOW ANSWER)

Option A provides no security. Option B relies on Snowflake's region IP addresses, which are not guaranteed and may change. Option C focuses on outbound traffic from the container service but doesn't properly restrict inbound access. Option D restricts outbound traffic unnecessarily and doesn't address inbound security. Option E is the MOST secure. It uses ingress and egress rules to control both inbound and outbound traffic for the container service and grants the necessary privileges to the service account. Ingress rule to specify the allowed inbound connection, and egress rule to only allow connection to snowflake's internal endpoint for stage access. This prevents other processes/users from accessing the stage.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 32

You are deploying a containerized application within a Snowflake Native App using Snowpark Container Services (SPCS). This application needs to interact with Snowflake's metadata, specifically retrieving information about tables and views within the customer's account. Which of the following methods is the MOST appropriate and secure way to achieve this?

- A.** Using the 'snowflake-jdbc' driver directly within the container, configured with the customer's Snowflake account credentials, which are passed as environment variables.
- B.** Leveraging the Snowpark Python API within the container to execute SQL queries against the Snowflake INFORMATION_SCHEMA views.
- C.** Creating a REST endpoint within the container that accepts SQL queries as input and executes them against Snowflake using the application's internal service account, then returns the results as JSON.
- D.** Using the Snowflake CLI (snowsql) within the container, configured with the application's service account credentials, to execute queries against the INFORMATION_SCHEMA.
- E.** Exposing a Snowflake UDF as an API endpoint that the container can call to retrieve metadata. The UDF executes the SQL queries against the INFORMATION_SCHEMA.

Answer: ([SHOW ANSWER](#))

Option B is the most appropriate and secure method. Snowpark Python API provides a secure and managed way to interact with Snowflake, leveraging the application's service account without exposing sensitive credentials directly within the container or requiring direct JDBC connections. Option A is highly insecure due to exposing credentials. Option C is viable, but requires extra management and maintenance of REST endpoint. Option D is possible but less secure than B for native apps. Option E adds unnecessary complexity.

NEW QUESTION: 33

Consider the following SQL code snippet intended to be part of a Snowflake Native Application. This application aims to provide data quality checks. Identify potential security vulnerabilities and recommend secure coding practices to mitigate them. Assume the application uses user-supplied input for filtering the data.

- A.** The code is safe because Snowflake automatically sanitizes all user input.
- B.** Use parameterized queries (prepared statements) or the 'SYSTEM\$SQL_INJECTION_CHECK' function to prevent SQL injection vulnerabilities.
- C.** Implement strict input validation, including data type checks and limiting the length and format of user-supplied input, before incorporating the input into the SQL query. Use Javascript functions like 'encodeURIComponent'.
- D.** Enclose the entire SQL query in a 'TRY...CATCH' block to prevent errors from propagating and potentially revealing sensitive information.
- E.** Sanitize input by escaping special characters using a regular expression, then concatenate the sanitized input directly into the SQL query.

Answer: ([SHOW ANSWER](#))

Option B and C are correct: Parameterized queries prevent SQL injection because the database treats the input as data, not as executable code. Strict input validation limits the possibility of malicious input being used to alter the SQL query's intent. Option A is incorrect since Snowflake does not automatically sanitize user input. Option D is beneficial for error handling but does not prevent SQL injection. Option E is risky, regular expressions are not guaranteed to catch all injection attempts.

NEW QUESTION: 34

You are developing a Snowflake Native App that needs to be deployed across multiple regions. Your application relies on data stored in internal stages. Considering the limitations of Snowflake Native App Framework regarding regional availability and data residency, which of the following strategies is the MOST appropriate to ensure data availability and minimize latency for users in different regions?

- A.** Replicate the entire Snowflake account across all regions where the app is deployed.
- B.** Utilize Snowflake's data replication feature to replicate the internal stages and associated data to regional Snowflake accounts.
- C.** Package all necessary data within the application itself, eliminating the need for external data sources.
- D.** Configure the application to access data in a central, globally accessible stage, regardless of the user's region.
- E.** Design the application to use external stages (e.g., AWS S3, Azure Blob Storage) in each region and implement logic to access the nearest stage based on the user's location.

Answer: ([SHOW ANSWER](#))

The best approaches are B and E. Replicating internal stages (B) keeps data close to users in each region. Using regional external stages (E) with location-aware logic offers flexibility and potentially lower latency. Replicating the entire account (A) is overkill and cost-prohibitive. Packaging data within the app (C) is impractical for large datasets and makes updates difficult. Using a central stage (D) negates the benefits of regional deployment. External stages are preferred because Native Apps can only read/write to internal stages within the Application package and not in the consumer's account.

NEW QUESTION: 35

You are developing a Snowflake Native App that requires accessing a specific table within the consumer's account. The consumer should not have direct access to this table. Which of the following is the MOST secure and appropriate method to grant the necessary access?

- A.** Granting the CONSUMPTION usage privilege on the provider account database to the consumer account, and then granting SELECT on the specific table to a role that the app assumes.

- B.** Creating a Snowflake Secure View on the table in the provider account and sharing this view with the consumer account using Data Sharing.
- C.** Creating a Snowflake Secure UDF that queries the table within the provider account and exposing the UDF to the consumer account through a package.
- D.** Granting direct SELECT access to the consumer role on the table in the provider account, ensuring to revoke access after the app is installed.
- E.** Employing a Stored Procedure owned by the application role that queries the table in the provider account, and allowing access to the procedure via API integration.

Answer: ([SHOW ANSWER](#))

Secure UDFs allow you to execute queries in the provider account without exposing the underlying table directly to the consumer. This isolates the data and minimizes the risk of unauthorized access. Stored procedures offer similar functionality, but UDFs are generally simpler and more performant for basic data retrieval. Secure Views via data sharing expose the structure to the consumer account, and granting direct access is a major security risk.

NEW QUESTION: 36

Within a Snowflake Native Application, you need to implement a feature that is only available in the latest version of the application, and the usage of that feature needs to be tracked. How would you accomplish this within the application code, considering that you want to avoid exposing internal details about version numbers directly to the consumer?

- A.** Use the 'APPLICATION\$VERSION' function within a stored procedure to conditionally enable the feature and log the usage to a separate table. Make sure the logs are maintained within the producer account.
- B.** Create a separate role that is only granted to users of the latest version, and use this role to control access to the feature. Use a view to abstract the role requirement.
- C.** Implement a UDF (User-Defined Function) that checks the application version and returns a flag indicating whether the feature is available. Use this flag in SQL queries to conditionally enable the feature and track its usage.
- D.** Use Snowflake's Resource Monitors to track resource usage by different versions of the application.
- E.** Rely on the consumer to explicitly enable the feature through a configuration parameter, then log the feature's usage.

Answer: ([SHOW ANSWER](#))

The correct answer is (A). Using 'APPLICATION\$VERSION' within a stored procedure allows you to conditionally enable the feature based on the application version. Logging the usage to a separate table (within the producer account) provides valuable insights into feature adoption without exposing version details to the consumer. Resource monitors (D) track resource consumption but not specific feature usage. Roles (B) can control access but don't directly provide usage tracking. Option E relies too much on consumer action. While UDFs (C) can check the version, doing so in SQL queries for conditional enabling might not be the most performant or maintainable approach. All logs should be captured within Producer account.

NEW QUESTION: 37

You are developing a Snowflake Native Application that utilizes a Streamlit frontend. The application requires access to specific data in the consumer account. You've already defined the necessary roles in your provider account. Which of the following steps are absolutely necessary to grant the Streamlit application the appropriate permissions to access the consumer's data? Assume the provider account has already shared the application package.

- A.** In the consumer account, grant the 'IMPORTED PRIVILEGES' privilege on the application package to a role that Streamlit will assume. Ensure the Streamlit application has code to execute 'snowflake.connector.connect()' with the correct consumer account credentials.
- B.** In the consumer account, create a new application role and grant the necessary privileges (e.g., 'SELECT on tables') to this role. In the Streamlit code, use `streamlit.experimental_get_query_params()` to pass the role name as a query parameter to the application.
- C.** Define an application role in the application package and grant the necessary privileges within the package installation script using 'GRANT APPLICATION ROLE TO SHARE'. In the consumer account, grant the application role to a user. No code changes are needed in the Streamlit application. Ensure to also grant the application role to the application in the install script with 'GRANT APPLICATION ROLE TO APPLICATION'.
- D.** Create a new database role in the provider account and grant the necessary privileges (e.g., 'SELECT on tables') to this role. Share this database role to the consumer account using secure data sharing. No code changes are needed in the Streamlit application.
- E.** Within the application package installation script, grant the necessary privileges (e.g., 'SELECT on tables') directly to the 'APPLICATION' object representing the installed application using 'GRANT ON TO APPLICATION'. In the consumer account, no further privilege grants are required. Also ensure to 'GRANT APPLICATION ROLE app_public TO APPLICATION'.

Answer: ([SHOW ANSWER](#))

Options C and E are the most accurate. Option C is correct because it leverages application roles, which are the recommended way to manage privileges in Snowflake Native Apps. Granting the application role to a user in the consumer account allows them to access the data via the application. Option E is correct because granting privileges directly to the 'APPLICATION' object is essential for the application to function correctly with its own access rights. Additionally, granting 'app_public' ensures basic functionality is available to the application. Option A is incorrect because applications should not be connecting to the database using standard connectors with explicit credentials in Native Apps. Option B is incorrect because passing role names as query parameters is not a secure or recommended practice. Option D is incorrect because sharing a database role directly is not the appropriate mechanism for granting access within a Snowflake Native App. The application package and its associated roles should manage the access.

NEW QUESTION: 38

Consider the following SQL code snippet within a Snowflake Native Application:

- A. The statement will execute successfully, but the 'session_id' will always be null as the 'READ SESSION' privilege does not apply within stored procedures.
- B. The statement will fail because the 'READ SESSION' privilege is not automatically granted to stored procedures. You must explicitly grant the privilege to the stored procedure's owner.
- C. The statement will execute successfully, returning the session ID of the user who installed the Native Application.
- D. The statement will execute successfully, returning the session ID of the current session in which the stored procedure is running, provided the caller has USAGE privilege on the database.
- E. The statement will execute successfully, returning the session ID of the current session in which the stored procedure is running, provided the Native App has the 'READ SESSION' privilege (implicitly or explicitly).

Answer: ([SHOW ANSWER](#))

The 'READ SESSION' privilege, if granted to the Native App (either implicitly or explicitly, depending on how it's managed within the app package), allows access to session-related information within the app's context. The session ID will correspond to the current execution context of the stored procedure. Options A and B are incorrect because 'READ SESSIONS' does apply within stored procedures if properly authorized. Option C is incorrect as Native app will work on their own context, not the end user context and Option D incorrect as only app owner can use this privilege.

NEW QUESTION: 39

You are building a Snowflake Native Application that includes a stored procedure to perform data transformation. This stored procedure needs to access external data sources using a secure API. Which of the following methods provides the MOST secure and recommended approach for managing API credentials within the Snowflake Native Application context?

- A. Store the API credentials directly within the stored procedure code as encrypted strings. Decrypt them at runtime using a key stored in a separate table.
- B. Store the API credentials in a configuration file included within the application package. Read the credentials from this file within the stored procedure.
- C. Use Snowflake's Secret object to securely store the API credentials. Access the Secret within the stored procedure using function.
- D. Use the consumer's account's environment variables to store API credentials. The application retrieves credentials from there.
- E. Hardcode the API credentials directly within the stored procedure code. This is the simplest approach for development and testing.

Answer: ([SHOW ANSWER](#))

Option C provides the most secure and recommended approach. Snowflake's Secret object is specifically designed for securely storing and managing sensitive information like API credentials. The 'SYSTEM \$GET_SECRET' function allows controlled access to these secrets within the stored procedure. Option A introduces unnecessary complexity and potential vulnerabilities with custom encryption. Option B exposes the credentials within the application package, making it less secure. Option D is not feasible and secure, because consumer managed secrets is not a built in functionality for Native apps. Option E is extremely insecure and should never be used.

NEW QUESTION: 40

Consider the following 'manifest.yml' file for a Snowflake Native App:

- A. The application will fail to install because the version property is missing. Each version must explicitly be specified in the manifest file.
- B. The application version will default to '0.1.0' and the application can be installed successfully by any consumer account.
- C. The application version will be automatically assigned by Snowflake during the installation process on the consumer side.
- D. The application will fail to install because no 'privileges' are listed in the manifest file. All applications must declare the necessary privileges.

E. The application version will default to '0.0.1'. If the application includes a view, then the view needs access to the user data and must have a privilege declared to allow users to view data from the application.

Answer: ([SHOW ANSWER](#))

A valid version is mandatory in 'manifest.yml' to install native apps. When version is missing, the installation will fail.

NEW QUESTION: 41

A Native Application developer, Alice, is building a data processing pipeline within her Snowflake Native App that includes a scheduled task. The task executes a stored procedure which needs to access data from a table owned by the consumer account. The application package version '1.0.0' has been released to a consumer. The task has been created and suspended. Which of the following SQL commands are required for Alice, as the application provider, to grant the necessary privileges on the consumer's environment to enable the task to execute successfully when it is resumed? Select ALL that apply.

- A. GRANT EXECUTE TASK ON ACCOUNT TO SHARE;
- B. GRANT EXECUTE TASK ON TASK my_task TO APPLICATION ROLE app_public;
- C. GRANT EXECUTE MANAGED TASK ON ACCOUNT TO SHARE;
- D. GRANT EXECUTE TASK ON TASK my_task TO APPLICATION ROLE app_owner;
- E. GRANT EXECUTE TASK ON DATABASE TO APPLICATION ROLE app_public;

Answer: ([SHOW ANSWER](#))

When managing tasks in Snowflake Native Apps that run in the consumer account, the 'EXECUTE MANAGED TASK' privilege on the account share is required to allow the application provider to manage and execute the tasks within the consumer's environment. Additionally, the 'EXECUTE TASK' privilege needs to be granted to the application role that is responsible for executing the task. This is typically 'app_public' which is given execute task privileges. 'app_owner' is typically only used for managing the application itself.

NEW QUESTION: 42

A Snowflake Native App developer encounters an issue where the application fails to install on a consumer's account due to insufficient privileges. The developer has verified that the necessary privileges are declared in the application. privileges section of the manifest file. However, the installation still fails. What could be the most likely cause of this issue, and how can it be resolved?

- A. The consumer's account is not on the same Snowflake region as the provider account. Verify and align the regions.
- B. The consumer's account does not have the SNOWFLAKE.APP_INSTALLER role enabled. Ensure this role is granted to the appropriate user.
- C. The privileges declared in the manifest file are not supported by the consumer's Snowflake edition. Review and adjust the required privileges.
- D. The consumer account has object level grants conflicting with the required grants. Revoke the object level grants and try again.
- E. The privileges were declared in the manifest file, but were not actually granted to the application role in the provider account before creating the version. Grant the declared privileges to the application role using GRANT ON ACCOUNT TO APPLICATION ROLE and create a new version.

Answer: ([SHOW ANSWER](#))

Even if the privileges are declared in the manifest file, they must be explicitly granted to the application role in the provider account before a new version is created. The consumer's Snowflake edition might not support all privileges (but the error would be slightly different), SNOWFLAKE.APP_INSTALLER role does not exist, regional difference does not matter as such if provider created version after granting the privileges. The application's role in provider account needs required permission using GRANT ON ACCOUNT TO APPLICATION ROLE.

NEW QUESTION: 43

You are designing a Snowflake Native App that relies on a specific version of a shared library. You want to ensure that your application continues to function correctly even if the provider of the shared library releases a new, incompatible version. How can you achieve version isolation for your application's dependencies within the Snowflake Native App Framework?

- A. Utilize the 'feature' to directly link your application to a specific version of the shared library hosted in an external repository.
- B. Package the required version of the shared library directly within your application's code bundle, ensuring it is isolated from any external updates.
- C. Leverage Snowflake's dependency management system to specify a version range for the shared library in your application's setup script.
- D. Implement a custom version compatibility check within your application code to gracefully handle potential incompatibilities with newer versions of the shared library.
- E. Utilize Snowflake's 'API_INTEGRATION' to create endpoint where you have version control for the shared library.

Answer: ([SHOW ANSWER](#))

Packaging the required version of the shared library directly within your application's code bundle (B) provides the most reliable version isolation. This ensures that your application always uses the version it was designed for, regardless of any external updates. While options A, C, D, and E are valid approaches in other contexts, they do not provide the same level of isolation within the Snowflake Native App Framework. Snowflake Native Apps do not utilize for this purpose, nor do they have a specific dependency management system like option C. Version checking (D) is a good practice but doesn't prevent the application from being affected by the newer version. Using SAPI INTEGRATION' would not provide version control.

NEW QUESTION: 44

You are designing a Streamlit application within a Snowflake Native App package. This application displays data from a customer's Snowflake account. The application needs to enforce strict data governance policies, ensuring that users only see the data they are authorized to view based on their roles and permissions within the customer's account. Which strategies and Snowflake features should be employed to achieve this?

- A. Implement row-level security policies directly within the Streamlit application using Python code to filter the data.
- B. Leverage Snowflake's row access policies to automatically filter data based on the user's role and context at the database level.
- C. Create separate Streamlit applications for each user role, each with its own data access configuration.
- D. Use Snowflake's masking policies to redact sensitive data based on the user's role and context at the database level.
- E. Use external tables to fetch the data.

Answer: ([SHOW ANSWER](#))

Row access policies (B) are the recommended approach for implementing row-level security in Snowflake, as they are applied at the database level and ensure consistent enforcement regardless of the application accessing the data. Masking policies (D) are crucial for redacting sensitive information based on user roles. Implementing data filtering within the Streamlit application (A) is less secure and harder to maintain. Creating separate applications for each role (C) is inefficient and not scalable. External tables are for accessing data outside of Snowflake not controlling access to existing data.

NEW QUESTION: 45

You are building a Snowflake Native Application that provides a data transformation service. You want to allow consumers to configure certain aspects of the transformation pipeline (e.g., the name of the target table, specific transformation logic). What are the recommended approach(es) to allow consumers to customize the application's behavior without directly modifying the application code?

- A. Require consumers to fork the application's repository and modify the SQL code directly.
- B. Expose parameters within the setup script and stored procedures that consumers can set during application installation or execution.
- C. Utilize Snowflake's Secure UDFs (User-Defined Functions) and allow consumers to provide their own UDFs as configuration.
- D. Implement a configuration table within the application's schema that consumers can update with their desired settings. The application code reads these settings at runtime.
- E. Include default configurations within a file on the external stage, which consumers are expected to download, modify, and then upload back as a different file for the application to load.

Answer: ([SHOW ANSWER](#))

Options B and D are correct. Exposing parameters in the setup script and stored procedures is a standard way to allow customization during installation and execution. A configuration table provides a flexible way for consumers to manage settings that can be read by the application. Option A is incorrect because forking the repository makes it difficult to manage updates and maintain security. Option C has security implications, as malicious code can be injected through user-defined UDFs. Option E is incorrect as the file download, modify, and upload method are complex for users to configure.

NEW QUESTION: 46

You are developing a Snowflake Native Application. As part of your application deployment process, you need to programmatically register a new version of your application package and promote it through the stages (e.g., 'dev', 'test', 'prod'). You are using the 'SYSTEM\$REGISTER_APPLICATION_PACKAGE_VERSION' function. Which of the following statements accurately describe valid arguments and usage considerations for this function?

- A. The 'EXTERNAL ID' must be a globally unique identifier for each version across all Snowflake accounts. If The it's not unique, the registration will fail.
- B. The 'SOURCE_PACKAGE' argument is optional, and if omitted, the function will automatically generate a new application package based on the current
- C. The 'VERSION' argument is a display name for the version and doesn't impact version ordering or compatibility checks during installation.
- D. The PATCH' argument is required, it should be incremented for minor bug fixes or security updates within the same VERSION.

E. The 'SOURCE_PACKAGE' must be an application package in the same account as the one where the new version is being registered.

Answer: ([SHOW ANSWER](#))

The 'EXTERNAL ID' must be globally unique. The 'PATCH' allows for minor bug fixes. The 'SOURCE PACKAGE' indicates the application to clone the version from, which should be in the same account. The 'VERSION' argument is not just a display name; it is part of the version identifier and has implications for compatibility. 'SOURCE_PACKAGE' is required. 'SYSTEM \$REGISTER_APPLICATION_PACKAGE_VERSION' takes these parameters, and requires package to be in the same account.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 47

You are packaging a Snowflake Native App using the 'snowflake nativeapp package' command. The app relies on a specific external Python library that isn't available by default in Snowflake. Which methods can you use to include this library in your application package and ensure it's accessible within your application's procedures and functions?

- A.** Upload the Python library as a stage object and reference it in the 'manifest.yml' file using the 'artifacts' section. Snowflake will automatically include it in the application package and make it available during runtime using the 'IMPORT' statement in the handler code.
- B.** Bundle the Python library as a ZIP file along with your application code. When creating the function using 'CREATE FUNCTION', specify the ZIP file as part of the 'IMPORTS' clause. Snowflake automatically extracts and makes the library available to the function.
- C.** Use the Anaconda channel integration. Specify the required package in the 'manifest.yml' file, and Snowflake automatically downloads and installs the library during application installation. This only works for packages available through Anaconda.
- D.** Combine options B and C. Use the Anaconda channel integration when possible, and for custom or unavailable packages, use the ZIP file import method for individual functions. Prioritize Anaconda for easier dependency management.
- E.** Upload the Python library using the 'PUT' command and explicitly reference it in the 'manifest.yml' under the 'artifacts' section, pointing to the internal stage location. Use the 'IMPORT' statement within your stored procedure or UDF handler code to make the library available during execution.

Answer: ([SHOW ANSWER](#))

The correct answer is D. Snowflake allows using both Anaconda channel for readily available packages and ZIP file imports for custom or unavailable ones. Utilizing Anaconda streamlines dependency management where possible, while the ZIP file method provides flexibility for including any Python library directly with the function/procedure deployment.

NEW QUESTION: 48

A Snowflake Native Application uses a task 'aggregate_data' to aggregate data. This task is owned by the 'app_public' role and executes a stored procedure 'aggregate_data_proc'. Consumers have reported the task sometimes fails intermittently, citing permission issues, specifically related to accessing a specific table in the consumer account. The application provider has granted 'SELECT' privilege on the table to the 'app_public' role. However, the failures persist. What is the MOST LIKELY cause of the intermittent failures related to executing the task?

- A.** The stored procedure is executed with definer's rights (OWNER), so the privilege check occurs against the application provider's account, not the consumer's account.
- B.** The stored procedure is executed with caller's rights (CALLER), but the application role 'app_public' doesn't have the ACTIVATION privilege on the task.
- C.** The task 'aggregate_data' is executing with definer's rights (OWNER) of the application, so the grant of SELECT privilege to 'app_public' is sufficient, and there is likely an issue with transient network connectivity.
- D.** The stored procedure is executed with caller's rights (CALLER), but the application role 'app_public' doesn't have sufficient warehouse privileges in the consumer account.
- E.** The stored procedure is executed with caller's rights (CALLER), and the 'EXECUTE TASK' privilege wasn't granted correctly, which causes the user to randomly lose access on the Snowflake UI.

Answer: ([SHOW ANSWER](#))

When a stored procedure is executed by a task, the stored procedure runs with the rights of the OWNER by default. This means that even if you grant 'SELECT privilege on the table to 'app_public' (the task's role), if the stored procedure is executing with definer's rights, it is checking the privileges in the application provider's account, not the consumer's. Therefore, the application provider's account needs to have SELECT privilege on the table. It is unlikely there is any issues related to warehouse rights, or that they would resolve transient network connectivity issues. The EXECUTE TASK' privilege only relates to being able to execute the task, not what the task can access.

NEW QUESTION: 49

You are designing a setup script for a Snowflake Native Application. This script needs to perform the following actions: 1. Create a secure view that exposes a subset of data from a table owned by the provider account. 2. Grant 'SELECT privilege on this secure view to a specific role within the consumer account, but ONLY if that role already exists in the consumer account. 3. Create a UDF in the provider account which calls external function. This UDF needs access to a secret stored in a Snowflake secret object. Which of the following code snippets BEST implements these requirements, ensuring both security and proper privilege management?

```
CREATE OR REPLACE SECURE VIEW my_secure_view AS SELECT  FROM provider_table WHERE some_condition = true;
GRANT SELECT ON my_secure_view TO ROLE consumer_role;
CREATE OR REPLACE SECURE VIEW my_secure_view AS SELECT  FROM provider_table WHERE some_condition = true;
BEGIN
  IF (SELECT 1 FROM INFORMATION_SCHEMA.ROLES WHERE ROLE_NAME = 'consumer_role') THEN
    GRANT SELECT ON my_secure_view TO ROLE consumer_role;
  END IF;
END;
CREATE OR REPLACE FUNCTION my_udf() RETURNS VARCHAR LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('requests')
SECRETS = ('MY_SECRET'=MY_SECRET_OBJECT)
HANDLER = 'main'
AS $$
import requests
def main():
    secret_value = requests.get('snowflake://secrets/MY_SECRET').text
    #Use secret value in UDF logic
    return 'success'
$$
```

B. ;

```

CREATE OR REPLACE SECURE VIEW my_secure_view AS SELECT  FROM provider_table WHERE some_condition = true;
CREATE OR REPLACE FUNCTION my_udf() RETURNS VARCHAR LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('requests')
HANDLER = 'main'
AS $$
import requests
def main():
    #Use secret value in UDF logic
    return 'success'
$$
;
GRANT SELECT ON my_secure_view TO ROLE consumer_role;

```

C.

```

CREATE OR REPLACE SECURE VIEW my_secure_view AS SELECT  FROM provider_table WHERE some_condition = true;
GRANT SELECT ON my_secure_view TO ROLE consumer_role;
CREATE OR REPLACE FUNCTION my_udf() RETURNS VARCHAR LANGUAGE PYTHON
RUNTIME_VERSION = '3.8'
PACKAGES = ('requests')
HANDLER = 'main'
AS $$
import requests
def main():
    #Use secret value in UDF logic
    return 'success'
$$
;

```

D.

```
CREATE OR REPLACE SECURE VIEW my_secure_view AS SELECT  FROM provider_table WHERE some_condition = true;
BEGIN
  IF (SELECT 1 FROM INFORMATION_SCHEMA.ROLES WHERE ROLE_NAME = 'consumer_role') THEN
    GRANT SELECT ON my_secure_view TO ROLE consumer_role;
  END IF;
END;
CREATE OR REPLACE FUNCTION my_udf() RETURNS VARCHAR
RETURNS NULL ON NULL INPUT
VOLATILE
LANGUAGE JAVASCRIPT
AS
$$
  // Retrieve secret from Snowflake secret object
  var secret_value = snowflake.secrets.get('MY_SECRET');
  // Call the external API with the secret value
  var request = require('request');
  var response = request('GET', 'https://external-api.example.com?key=' + secret_value);
  return response.body;
$$
;
```

E.

Answer: (SHOW ANSWER)

Option B correctly implements all requirements. It uses a 'BEGIN...ENDS' block with an statement to check for the existence of the 'consumer_role' before granting the privilege. This prevents errors if the role doesn't exist. Also it creates a UDF which utilizes Snowflake Secret object to fetch secretes. Using Javascript code inside UDF directly to fetch secrets is not allowed. Options A, C and D do not check for role existence and doesn't configure Snowflake secret object

NEW QUESTION: 50

You are developing a Snowflake Native Application and need to track changes made to the application code across different versions. You want to leverage Snowflake's built-in capabilities for version control and auditing. Which of the following approaches is the MOST suitable and efficient for achieving this?

- A. Manually maintain a change log in a separate Snowflake table, documenting each code modification with associated version numbers and timestamps.
- B. Integrate with an external version control system (e.g., Git) and store application code and deployment scripts in the repository. Use CI/CD pipelines to manage deployments to Snowflake.
- C. Use Snowflake's Time Travel feature to query historical versions of application code objects (e.g., stored procedures, UDFs) and compare differences between versions.
- D. Use Snowflake's Streams feature to capture changes made to application code objects and store them in a separate change table for auditing.
- E. Utilize Snowflake's Information Schema views (e.g., 'INFORMATION_SCHEMA.PROCEDURES', 'INFORMATION_SCHEMA.FUNCTIONS') in conjunction with Time Travel to query historical metadata and code definitions.

Answer: (SHOW ANSWER)

Option E is the most suitable and efficient solution. Combining Snowflake's Information Schema views with Time Travel provides a comprehensive and native way to track changes to application code objects. The Information Schema provides metadata about database objects, and Time Travel allows you to query historical versions of those objects. This approach leverages Snowflake's built-in capabilities without requiring external tools or manual change logging. Options A and B requires extra configuration of the whole process. Option C while feasible, Time Travel has to be enabled on each component, which makes option E superior. Option D is designed for data and not code auditing.

NEW QUESTION: 51

A software vendor is developing a Snowflake Native Application for fraud detection. They plan to distribute it via the Snowflake Marketplace. They have implemented robust data governance policies within their application. What steps must they take to ensure that these policies are enforced and clearly communicated to consumers using the application in their own Snowflake accounts, specifically related to data masking policies?

- A.** Data governance policies are automatically inherited by consumer accounts when they install the application.
- B.** The vendor should create a separate documentation detailing the data governance policies and provide it alongside the Snowflake Marketplace listing. Consumers are responsible for implementing these policies in their accounts.
- C.** The vendor should create masking policies. Policies created with the 'OWNERSHIP' clause will be granted to the application database role. If the application creates and then shares tables, the table owner (database role) will have the ability to grant 'APPLY MASKING POLICY' on those objects.
- D.** The vendor must create all masking policies using the WITH MASKING POLICY clause to ensure policies are enforced on the consumer side.
- E.** Data governance policies should be implemented within the application's UI and enforced programmatically through stored procedures.

Answer: C ([LEAVE A REPLY](#))

Masking policies with the 'OWNERSHIP' clause allow the database role to grant 'APPLY MASKING POLICY' on objects created and shared by the application. This grants the necessary permissions to enforce masking policies on the consumer's end. Data policies are not automatically inherited, and consumers shouldn't solely be responsible for implementing policies themselves.

NEW QUESTION: 52

You are developing a Snowflake Native Application that uses Streamlit to present data. The application relies on a UDF to perform calculations. The UDF resides in the application's schema within the container. The application package is named and the application itself is named 'my_app'. You need to grant the application the necessary privileges to execute this UDF. Which of the following SQL statements, executed by the application provider, will correctly grant the required privilege to the application?

- A.** `GRANT USAGE ON FUNCTION my_app_package.application.my_udf(VARCHAR) TO APPLICATION ROLE app_public;`
- B.** `GRANT EXECUTE ON FUNCTION my_app_package.application.my_udf(VARCHAR) TO APPLICATION ROLE app_public;`
- C.** `GRANT USAGE ON FUNCTION my_app_package.application.my_udf(VARCHAR) TO APPLICATION my_app;`
- D.** `GRANT EXECUTE ON FUNCTION my_app_package.application.my_udf(VARCHAR) TO APPLICATION my_app;`
- E.** `GRANT USAGE ON FUNCTION my_app_package.application.my_udf(VARCHAR) TO APPLICATION ROLE app_role;`

Answer: ([SHOW ANSWER](#)**)**

The correct answer is B. The 'EXECUTE' privilege is required for an application to execute a UDF. Additionally, 'APPLICATION ROLE app_public' is the correct role to grant the privilege to, enabling all users within the application to execute the function. Option A is incorrect because USAGE is not sufficient for executing a UDF. Options C and D are incorrect because you should grant permissions to an APPLICATION ROLE, not the Application directly. Option E is incorrect because 'app_role' is a generic term, and 'app_public' should be used to define the public Application role.

NEW QUESTION: 53

A Snowflake Native App developer is building an application package that contains multiple versions. They want to ensure that a consumer upgrading from version 1.0 to version 2.0 MUST execute a specific set of SQL scripts as part of the upgrade process to migrate existing application data. Which Snowflake Native App feature should they leverage to achieve this?

- A.** Application Role-Based Access Control (RBAC)
- B.** A post-install script defined in the setup script.
- C.** A version change handler defined within the application package.
- D.** Snowflake Streams and Tasks to monitor the application version and trigger the migration scripts automatically.
- E.** Snowflake Pipes to load and transform the existing data.

Answer: ([SHOW ANSWER](#)**)**

Option C is the correct approach. A version change handler is a dedicated script that runs automatically when a consumer upgrades or downgrades the application. This handler can contain the necessary SQL scripts to migrate data, update schemas, or perform any other actions required during the version transition. Option B executes only during initial installation, not during upgrades. Option A, RBAC, manages permissions but doesn't execute scripts. Option D, Streams and Tasks, could be used, but it's more complex than a version change handler. Option E is used for continuous data loading.

NEW QUESTION: 54

You are designing the 'privileges.sqr' file for a Snowflake Native App. The app requires the ability to create and manage internal stages within the consumer's account to store temporary data. Which of the following SQL statements in 'privileges.sqr' would correctly grant the necessary privileges to the application role (assuming the application role is named 'APP_ROLE')?

- A. `GRANT USAGE ON WAREHOUSE TO ROLE APP_ROLE;`
- B. `GRANT CREATE STAGE ON DATABASE TO ROLE APP_ROLE;`
- C. `GRANT ALL PRIVILEGES ON DATABASE TO ROLE APP_ROLE;`
- D. `GRANT CREATE STAGE ON SCHEMA TO ROLE APP_ROLE; GRANT USAGE ON SCHEMA TO ROLE APP_ROLE;`
- E. `GRANT OWNERSHIP ON DATABASE TO ROLE APP_ROLE;`

Answer: D ([LEAVE A REPLY](#))

To create stages, the application role needs the 'CREATE STAGE' privilege. This privilege needs to be granted on the schema where the stages will be created, and usage privilege is also needed. Granting it at the database level is too broad (B, C, E), and only granting warehouse usage (A) doesn't address the requirement of creating stages.

NEW QUESTION: 55

A data analytics company, 'Data Insights Pro', is developing a Snowflake Native App that visualizes customer behavior data. The app needs to read data from a secure customer table provided by the consumer and write aggregate reports back to a separate consumer-owned table for compliance. The customer table contains sensitive PII. Which of the following combinations of application privileges and app capabilities are NECESSARY and SUFFICIENT to achieve this securely, ensuring minimal access is granted?

- A. APPLICATION role WITH OWNERSHIP on the REPORT_TABLE, APPLICATION role WITH SELECT privilege on the CUSTOMER_TABLE through a SECURE VIEW, and the 'network access' capability.
- B. APPLICATION role WITH INSERT privilege on the REPORT_TABLE, APPLICATION role WITH SELECT privilege on the CUSTOMER_TABLE through a SECURE VIEW, and the 'external network access' capability.
- C. APPLICATION role WITH INSERT privilege on the REPORT_TABLE, APPLICATION role WITH SELECT privilege on the CUSTOMER_TABLE, and the 'network access' capability.
- D. APPLICATION role WITH OWNERSHIP on the REPORT_TABLE, APPLICATION role WITH SELECT privilege on the CUSTOMER_TABLE, and the 'external network access' capability.
- E. APPLICATION role WITH INSERT privilege on the REPORT_TABLE, APPLICATION role WITH SELECT privilege on the CUSTOMER_TABLE through a SECURE VIEW, and no additional capabilities.

Answer: ([SHOW ANSWER](#)**)**

The app needs to write to REPORT_TABLE (requires INSERT, not OWNERSHIP as ownership implies broader privileges than required), read from CUSTOMER_TABLE (requires SELECT, and using a secure view is the best practice to minimize PII exposure), and does not require network access since everything happens within Snowflake's environment. No external network access is required as no data is transmitted externally.

NEW QUESTION: 56

Consider the following scenario: You are tasked with designing a setup script for a Snowflake Native Application. This script needs to perform several tasks including creating a database role, granting privileges to the role, and creating a schema. Your application also provides a configuration table that needs to be populated during the setup process. Which of the following set of statements are true regarding the order of operations and security best practices within the setup script? (Choose all that apply)

- A. It is best practice to create the database role before creating the schema, to immediately grant ownership on the schema to the created role.
- B. When granting privileges to the database role, use the 'APPLICATION' keyword to grant privileges on application objects, enhancing security and control.
- C. Populating the configuration table should be done before creating any other objects, as other parts of the application might rely on the configuration during object creation.
- D. The setup script should always be designed to be idempotent, meaning it can be executed multiple times without causing errors or unintended side effects. Use 'CREATE ... IF NOT EXISTS' where appropriate.

E. Setup Scripts should be designed in a way that it requires multiple manual interventions during execution.

Answer: ([SHOW ANSWER](#))

Option A is correct because creating the role first allows for immediate ownership and control over the newly created schema. Option B is correct as using the 'APPLICATIONS' keyword when granting privileges is a key security best practice in Snowflake Native Apps. It allows for controlled access to the data within the Application. Option D is correct because idempotency ensures the setup process can be retried if interrupted without causing issues. Option C is incorrect because, While application configuration is important, object creation, like defining schemas and roles usually must precede loading configuration, and depend on the existence of these core components. Option E is wrong since It should require minimal to zero manual intervention.

NEW QUESTION: 57

You are developing a complex Snowflake Native Application with multiple Python modules and dependencies. You've carefully crafted your 'environment.yml' file, but during the installation process in a consumer account, you encounter intermittent errors related to package conflicts. These errors are difficult to reproduce consistently. What is the MOST effective strategy, encompassing both design and troubleshooting, to mitigate these dependency resolution issues and ensure a reliable installation experience for your users? Select TWO options.

A. Use a broad range of package versions in your 'environment.yml' file to allow for flexibility during dependency resolution, reducing the likelihood of conflicts.

B. Vendor all third-party Python dependencies directly into your application package (i.e., include the package code directly instead of relying on Conda installation). This creates a self-contained application with no external dependencies.

C. Examine the Snowflake logs ('SYSTEM\$GET_PREDECESSORS' and 'SYSTEM\$GET_DEPENDENTS') for any other applications installed in the consumer account that might be conflicting with your application's dependencies.

D. Pin all dependencies to specific versions in your 'environment.yml' file, including transitive dependencies, to create a deterministic and reproducible environment.

E. Implement robust error handling in your Python modules to catch potential 'ImportError' exceptions and provide informative error messages to the user, even if the underlying cause is a dependency conflict.

Answer: ([SHOW ANSWER](#))

Options C and D provide the most effective combination of preventative measures and troubleshooting strategies. Option D (Pinning Dependencies): Pinning dependencies to specific versions in the 'environment.yml' file is crucial for creating a deterministic and reproducible environment. This ensures that the exact same versions of all packages are installed in every consumer account, eliminating version-related conflicts. While pinning can sometimes introduce challenges with upgrades, it provides stability and predictability during installation. Option C (Examining Snowflake Logs): Examining the Snowflake logs (using 'SYSTEM\$GET_PREDECESSORS' and 'SYSTEM\$GET_DEPENDENTS') can reveal if other applications installed in the consumer account are using conflicting versions of the same dependencies. This allows you to identify the source of the conflicts and potentially work with the other application provider to resolve the issue or adjust your own application's dependencies if possible. Option A is incorrect because using a broad range of package versions increases the likelihood of conflicts, not reduces it. Option B is a valid strategy, but it greatly increases the size and complexity of the application package. This would make the app less manageable and harder to upgrade or maintain. Option E is important for user experience, but it doesn't address the underlying dependency conflict. It's a good practice to include such error handling, but it's not a primary solution for dependency resolution issues.

NEW QUESTION: 58

You are developing a Snowflake Native Application that needs to track usage metrics within consumer accounts. Specifically, you want to log the number of times a particular function within your application is called, and also store certain operational statistics. Which of the following storage options are available to you within the application package to persist these metrics, and what are their implications? (Select TWO)

A. Using temporary tables within the application. These tables are automatically dropped at the end of the session.

B. Creating permanent tables within the application package using the 'APPLICATION' schema. These tables persist across installations and updates.

C. Leveraging external stages (e.g., AWS S3) to store the metrics. The application would need the necessary IAM roles and stage configuration.

D. Utilizing User-Defined Functions (UDFs) with external network access to push metrics to an external logging service.

E. Using the 'APPLICATION' schema to create persistent internal stages. Data written to these stages will persist across installations and updates.

Answer: ([SHOW ANSWER](#))

Options B and E are the correct choices. The 'APPLICATION' schema allows for the creation of permanent tables within the application package. These tables persist across installations and updates, making them suitable for storing usage metrics. Similarly internal stages created using the 'APPLICATION' schema will persist data across versions. Option A is incorrect as temporary tables are session-based.

Option C introduces external dependencies that are not typically ideal for tightly integrated Native Apps for internal metrics. Option D involves external network calls which might be restricted and less efficient for logging internal metrics.

NEW QUESTION: 59

You are developing a Snowflake Native Application that provides a data enrichment service. This service requires the consumer to grant the application access to a specific table in their account. You want to ensure that upgrades to your application do not inadvertently break existing consumer configurations. You decide to use a versioned schema. Which of the following steps are essential during the setup script execution to achieve this and maintain backward compatibility during future upgrades?

- A. Create a new unversioned schema named 'APPLICATION' and install all application objects within it. This provides a consistent namespace for all consumers, regardless of the application version.
- B. Create a versioned schema using 'CREATE SCHEMA VERSION = ' '. Grant the application role 'USAGE' on this versioned schema, and ensure all application logic references objects within this schema.
- C. Grant the application role 'OWNERSHIP' on the consumer's table directly. This allows the application full control and ensures consistent data access regardless of future schema changes within the application.
- D. Create a secure view in the versioned schema that accesses the consumer's table. Grant the application role 'SELECT' on this secure view. This provides a stable interface that isolates the application from direct changes to the consumer's table.
- E. Define a custom role within the application package and grant it the necessary privileges to read the consumer's table via a secure view. Then, grant the application role the USAGE privilege on this custom role. This ensures that the application role can assume the custom role's privileges for data access.

Answer: (SHOW ANSWER)

Options B and D are correct. Creating a versioned schema allows for managing changes in your application's data structures. By creating a secure view and granting the application role 'SELECT' on it, you isolate your application from direct modifications to the consumer's table, ensuring backward compatibility. Option A is incorrect because an unversioned schema doesn't address the need for backward compatibility during upgrades. Option C is generally discouraged due to security concerns of granting ownership. Option E, while useful, is not the core reason to use a versioned schema to protect from consumer changes.

NEW QUESTION: 60

An application developer is creating a Snowflake Native App that uses a user-defined function (UDF) 'MY_UDF' in 'APP_SCHEMA'. The UDF reads data from a table 'APP_TABLE' which is also in 'APP_SCHEMA'. The application role is 'APP_ROLE'. Which of the following 'GRANT' statements are necessary for the application to function correctly within the provider account, assuming the UDF is defined with 'VOLATILE' behavior and is using the caller's rights (i.e., 'EXECUTE AS CALLER')?

- ☐ GRANT USAGE ON DATABASE APP_DB TO ROLE APP_ROLE; GRANT USAGE ON SCHEMA APP_DB.APP_SCHEMA TO ROLE APP_ROLE; GRANT EXECUTE ON FUNCTION APP_DB.APP_SCHEMA.MY_UDF(NUMBER) TO ROLE APP_ROLE; GRANT SELECT ON TABLE APP_DB.APP_SCHEMA.APP_TABLE TO ROLE APP_ROLE;
- A. ☐ GRANT USAGE ON DATABASE APP_DB TO ROLE APP_ROLE; GRANT USAGE ON SCHEMA APP_DB.APP_SCHEMA TO ROLE APP_ROLE; GRANT EXECUTE ON FUNCTION APP_DB.APP_SCHEMA.MY_UDF(NUMBER) TO ROLE APP_ROLE;
- B. ☐ GRANT OWNERSHIP ON DATABASE APP_DB TO ROLE APP_ROLE; GRANT OWNERSHIP ON SCHEMA APP_DB.APP_SCHEMA TO ROLE APP_ROLE; GRANT OWNERSHIP ON FUNCTION APP_DB.APP_SCHEMA.MY_UDF(NUMBER) TO ROLE APP_ROLE; GRANT OWNERSHIP ON TABLE APP_DB.APP_SCHEMA.APP_TABLE TO ROLE APP_ROLE;
- C. ☐ GRANT ALL PRIVILEGES ON DATABASE APP_DB TO ROLE APP_ROLE; GRANT ALL PRIVILEGES ON SCHEMA APP_DB.APP_SCHEMA TO ROLE APP_ROLE; GRANT ALL PRIVILEGES ON FUNCTION APP_DB.APP_SCHEMA.MY_UDF(NUMBER) TO ROLE APP_ROLE; GRANT ALL PRIVILEGES ON TABLE APP_DB.APP_SCHEMA.APP_TABLE TO ROLE APP_ROLE;
- D. ☐ GRANT ALL PRIVILEGES ON DATABASE APP_DB TO ROLE APP_ROLE; GRANT ALL PRIVILEGES ON SCHEMA APP_DB.APP_SCHEMA TO ROLE APP_ROLE; GRANT ALL PRIVILEGES ON FUNCTION APP_DB.APP_SCHEMA.MY_UDF(NUMBER) TO ROLE APP_ROLE; GRANT ALL PRIVILEGES ON TABLE APP_DB.APP_SCHEMA.APP_TABLE TO ROLE APP_ROLE;

- ☐ GRANT USAGE ON DATABASE APP_DB TO ROLE APP_ROLE; GRANT USAGE ON SCHEMA APP_DB.APP_SCHEMA TO ROLE APP_ROLE; GRANT EXECUTE ON FUNCTION APP_DB.APP_SCHEMA.MY_UDF(NUMBER) TO ROLE APP_ROLE;
- E.

Answer: A ([LEAVE A REPLY](#))

The SAPP ROLE' needs 'USAGE' on the database and schema, 'EXECUTE' on the UDF, and 'SELECT' on the table the UDF reads from because the UDF executes as the caller and the application needs those permissions within the provider account. 'OWNERSHIP' and 'ALL PRIVILEGES' are not necessary, and generally bad practice for least-privilege security.

NEW QUESTION: 61

You are developing a Snowflake Native Application that requires access to a specific stage in the consumer's account to read configuration files. The application needs to be able to create temporary files in the same stage during its execution. Which set of privileges needs to be granted on the stage to the application's service account, and how do you grant them?

- A. Grant READ and WRITE privileges on the stage to the application's service account using the 'GRANT USAGE ON INTEGRATION TO APPLICATION' command.
- B. Grant USAGE and READ privileges on the stage and CREATE FILE on the schema containing the stage to the application's service account using Snowsight.
- C. Grant READ and WRITE privileges on the stage using Snowsight, and ensure the application uses a service integration with appropriate permissions.
- D. Grant USAGE on the database, USAGE on the schema containing the stage, and READ and WRITE privileges on the stage to the application's service account using a combination of SQL commands and Snowsight.
- E. Grant OWNERSHIP on the stage to the application's service account. No other privileges are required.

Answer: ([SHOW ANSWER](#))

To allow the application to read configuration files and create temporary files in the stage, it needs the following privileges: USAGE on the database, USAGE on the schema containing the stage, READ privilege to read existing files, and WRITE privilege to create new files. These can be granted through a combination of SQL commands (for database and schema USAGE) and Snowsight (for stage READ and WRITE). Option D correctly specifies all the necessary privileges and the method for granting them. Ownership (Option E) is not necessary and would grant excessive permissions. USAGE on integration is specific to external functions/API integrations, not stage access directly.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 62

A data science team within your organization is developing a Snowflake Native App that needs to access and process customer data stored in a Snowflake database. The app must only access data for customers residing in the 'USA' region. The team has created a custom role 'APP_ROLE' for the app to use. Which of the following is the MOST secure and efficient way to grant the app the necessary permissions to read the filtered data? Assume a secure view approach is preferred over direct table access.

- A. Grant the 'APP_ROLE' the 'SELECT' privilege on the base customer data table and rely on row-level security policies defined on that table.
- B. Create a secure view that filters customer data to include only customers in the 'USA' region and grant the SAPP_ROLE the 'SELECT' privilege on the secure view.
- C. Grant the SAPP_ROLE the 'OWNERSHIP' privilege on the base customer data table and implement data filtering logic within the app itself.
- D. Create a standard view (non-secure) that filters customer data to include only customers in the 'USA' region and grant the 'APP_ROLE' the 'SELECT' privilege on this view.
- E. Grant the 'APP_ROLE' the 'USAGE' privilege on the database and schema containing the customer data table. Data filtering should still be handled by the app.

Answer: ([SHOW ANSWER](#))

Secure views offer a crucial security advantage: they hide the underlying table structure and any associated metadata from the app, only exposing the filtered data. Row-level security (RLS) can work (A), but secure views provide an extra layer of abstraction and security, hiding the existence of other rows. Granting ownership (C) is highly insecure and unnecessary. Non-secure views (D) do not offer the security benefits required for sensitive data. USAGE privilege alone (E) doesn't grant any data access without SELECT.

NEW QUESTION: 63

You are developing a Snowflake Native Application using Python that processes financial transactions. The application relies on external stages for loading transaction data. After deploying the application to a consumer account, you observe that the application fails to load data from the stage, resulting in an 'Invalid stage name' error in the consumer's logs. Assume the stage was created and shared correctly.

Which of the following steps would BEST help you diagnose the problem and ensure the application works as expected?

- A.** Verify that the application manifest includes the correct stage definition with the 'privileges' parameter set to 'USAGE' on the stage object. Also, within your Python code, use the 'snowflake.connector' to explicitly grant USAGE privilege from the application role to the stage.
- B.** Ensure the stage definition in the 'manifest.yml' file uses the fully qualified name of the stage (database.schema.stage_name), and verify that the consumer has imported the stage's database and schema into their environment. In Python, use the 'DESC STAGE' command via a Snowflake connector to confirm existence and permissions.
- C.** Redeploy the application with the 'REPLACE' flag to automatically update the stage definition on the consumer side, and confirm that the application has sufficient privileges to access the storage integration associated with the stage. In Python, trap any 'Invalid stage name' exceptions and retry after a short delay.
- D.** Check the application's deployed version details in the consumer account's Snowflake UI to confirm the 'stage_name' parameter within the application manifest is defined correctly, and then within the Python code, make sure to utilize a fully qualified name when referring to the stage. Use Snowflake logging to write stage information to the event table.
- E.** Review the stage properties in both the provider and consumer accounts to ensure the storage integration is correctly configured and accessible from both sides. In the Python code, use the USE ROLE command to explicitly switch to the application's role before accessing the stage.

Answer: (SHOW ANSWER)

The BEST approach is to verify the stage definition in the 'manifest.yml' and ensure its deployed version details in the consumer account. Using fully qualified names in both the manifest and Python code is essential for the consumer to correctly resolve the stage. Correctly utilize Snowflake logging by writing stage information to the event table. This ensures the application can properly access the stage within the consumer's environment and provides debugging information if issues persist. Incorrect assumptions might lead to focusing on permissions when the fundamental issue is name resolution. Other options suggest using a 'snowflake.connector' to explicitly grant USAGE privilege from the application role to the stage or to utilize a fully qualified name when referring to the stage which will not solve the issue.

NEW QUESTION: 64

When designing the architecture of a Snowflake Native App, which of the following architectural considerations and trade-offs should you carefully evaluate to optimize performance, security, and cost-effectiveness? (Select all that apply)

- A.** Minimizing data transfer between the provider and consumer accounts by leveraging Snowpark and UDFs for data processing within the consumer's environment.
- B.** Storing all application data within the provider account and accessing it directly from the consumer account using data sharing to avoid data duplication.
- C.** Using secured API integration to call the external APIs to get real time details in the app to make the app more useful.
- D.** Implementing robust error handling and logging mechanisms to facilitate debugging and monitoring of the application's behavior in both the provider and consumer environments.
- E.** Choosing the appropriate data access model (e.g., UDFs, stored procedures, Snowpark) based on the specific use case and performance requirements, considering the overhead associated with each approach.

Answer: (SHOW ANSWER)

Options A, D, and E are crucial for optimal Snowflake Native App design. Minimizing data transfer (A) reduces costs and improves performance. Robust error handling and logging (D) are essential for debugging and monitoring. Choosing the correct data access model (E) is a key performance factor. Option B is incorrect because storing all data in the provider account and sharing to the consumer introduces unnecessary dependencies and potential security risks. Consumers also need to pay for querying your shared data. Option C is not always desirable, since calling external API every time would be too expensive.

NEW QUESTION: 65

You are developing a Snowflake Native Application that provides data enrichment services. You want to implement a phased rollout of a new version to your users. You have three distinct user groups: 'EarlyAdopters', 'StandardUsers', and 'CriticalUsers'. You need to use release directives to manage the upgrade process, ensuring that 'EarlyAdopters' receive the new version first, followed by 'StandardUsers' after a week of monitoring, and finally 'CriticalUsers' two weeks after that. Which of the following approaches is the MOST effective way to achieve this using Snowflake Native App release directives?

- A.** Create separate application packages for each user group and manage versions independently within each package.

- B.** Use the 'ALTER APPLICATION' command with the 'STAGE VERSIONS' clause to explicitly assign specific versions to each user group based on their tag values. For example: 'ALTER APPLICATION my_app STAGE_VERSIONS version1 .1 FOR TAG user_group = 'EarlyAdopters';'
- C.** Use release directives within a single application package to control version availability based on the application role. Create separate roles for each user group and grant them appropriate privileges.
- D.** Implement a custom upgrade mechanism within your application logic, querying a configuration table to determine which version each user should receive. This table is updated based on a predefined schedule.
- E.** Use release directives with tag-based versioning. Tag each user account with a tag indicating their user group (e.g., 'user_group = 'EarlyAdopters)'). Then, use the 'STAGE_VERSIONS' clause in the 'ALTER APPLICATION' command to associate versions with these tags, specifying a different activation time for each tag. For example: 'ALTER APPLICATION my_app STAGE_VERSIONS version1 .1 FOR TAG user_group = 'EarlyAdopters' ACTIVATE_AT = '2024-01-01 00:00:00'; ALTER APPLICATION my_app STAGE_VERSIONS version1 .1 FOR TAG user_group = 'StandardUserS ACTIVATE_AT = '2024-01-08 00:00:00';'

Answer: ([SHOW ANSWER](#))

Option E is the most effective and recommended approach. Release directives, combined with tag-based versioning, provide a built-in mechanism for managing phased rollouts. This allows you to target specific user groups based on their tags and control the activation time for each version, ensuring a controlled and monitored upgrade process. Option A is not efficient as it increases management overhead. Option B is not correct since it does not support ACTIVATE_AT option. Option C will not allow gradual release as it is tied to application role. Option D needs to implement a custom upgrade logic. Snowflake tag-based versioning is the recommended option.

NEW QUESTION: 66

A Snowflake Native App developer is creating a new application database. They need to ensure that only the application itself and designated roles within the provider account can access the database. Which of the following CREATE DATABASE statements would BEST achieve this, adhering to best practices for application isolation and security?

- A.** CREATE DATABASE my_app_db;
- B.** CREATE DATABASE my_app_db MANAGED ACCESS;
- C.** CREATE DATABASE my_app_db WITH MANAGED ACCESS;
- D.** CREATE DATABASE my_app_db DATA_RETENTION_TIME_IN_DAYS=1;
- E.** CREATE DATABASE my_app_db MANAGED SCHEMA;

Answer: ([SHOW ANSWER](#))

The 'WITH MANAGED ACCESS' clause is crucial for enforcing managed access to the database. This ensures that only the database owner (application) and explicitly granted roles have access to the objects within the database, promoting better security and control in a provider account. 'MANAGED SCHEMAS' is a schema-level parameter, not a DATABASE-level parameter. Option A creates a database without managed access. Option B is syntactically incorrect. Option D is unrelated to access control. Option E is not the same as managed access.

NEW QUESTION: 67

You are designing a Snowflake Native Application that requires fine-grained access control to its data within the consumer account. You want to ensure that consumers can only access data relevant to their specific department. Which of the following Snowflake features, when combined, offer the MOST secure and flexible solution for implementing this requirement within the context of a Snowflake Native Application?

- A.** Views and Stored Procedures with Caller's Rights.
- B.** User-Defined Functions (UDFs) and Row Access Policies.
- C.** External Tables and Secure Views with Owner's Rights.
- D.** Row Access Policies and Secure Views with Caller's Rights.
- E.** Dynamic Data Masking policies and Views with Caller's Rights.

Answer: ([SHOW ANSWER](#))

The best combination is Row Access Policies and Secure Views with Caller's Rights. Row Access Policies: Allow you to filter data based on the user accessing it, ensuring they only see data relevant to their department. Policies are applied to tables. Secure Views with Caller's Rights: Allow you to encapsulate the data access logic, ensuring that the row access policies are always enforced, and that consumers

cannot bypass them. Caller's rights are essential here, as the access control decisions need to be made based on who is using the view. The other options are less suitable: A: Views and Stored Procedures can enforce access, but without Row Access Policies, they're less secure. B: UDFs can manipulate data but don't inherently enforce row-level security like Row Access Policies. C: External Tables are for data outside Snowflake; Owner's Rights are less secure than Caller's Rights in this scenario. E: Dynamic Data Masking hides data, it doesn't filter it, so it won't restrict access to specific rows.

NEW QUESTION: 68

You are developing a Snowflake Native App that requires granular access control to its underlying data for consumer accounts. You want to implement a secure pattern that limits direct access to the data while allowing specific operations through your app's procedures. Which of the following approaches represent the MOST secure and recommended practices regarding access control in the setup script, assuming you also want to grant USAGE on future schema?

A. Granting SELECT privilege directly on the underlying tables in the app's data schema to the consumer's role.

```
GRANT SELECT ON ALL TABLES IN SCHEMA app_db.data_schema TO APPLICATION ROLE app_role;
```

B. Creating secure views on top of the underlying tables and granting SELECT privilege on these views to the consumer's role, while ensuring consumers can only access the exposed view.

```
CREATE OR REPLACE SECURE VIEW app_db.public.secure_view AS SELECT * FROM app_db.data_schema.my_table; GRANT SELECT ON SECURE VIEW app_db.public.secure_view TO APPLICATION ROLE app_role;
```

C. Creating stored procedures with the 'EXECUTE AS CALLER' privilege and granting USAGE on the schema to the consumer's role. The stored procedures handle all data access logic.

```
CREATE OR REPLACE PROCEDURE app_db.public.get_data() RETURNS VARIANT LANGUAGE SQL EXECUTE AS CALLER AS $$
    SELECT * FROM app_db.data_schema.my_table;
```

```
$$; GRANT USAGE ON SCHEMA app_db.public TO APPLICATION ROLE app_role; GRANT EXECUTE ON PROCEDURE app_db.public.get_data() TO APPLICATION ROLE app_role;
```

D. Granting OWNERSHIP on the app's data schema to the consumer's role, allowing them full control over the data.

```
GRANT OWNERSHIP ON SCHEMA app_db.data_schema TO APPLICATION ROLE app_role;
```

E. Creating stored procedures with the 'EXECUTE AS OWNER' privilege and granting USAGE on the schema to the consumer's role. The stored procedures handle all data access logic.

```
CREATE OR REPLACE PROCEDURE app_db.public.get_data() RETURNS VARIANT LANGUAGE SQL EXECUTE AS OWNER AS $$
    SELECT * FROM app_db.data_schema.my_table;
```

```
$$; GRANT USAGE ON SCHEMA app_db.public TO APPLICATION ROLE app_role; GRANT EXECUTE ON PROCEDURE app_db.public.get_data() TO APPLICATION ROLE app_role;
```

Answer: (SHOW ANSWER)

Granting SELECT directly is insecure (A). Secure views provide some abstraction but still allow direct SELECT (B). Granting OWNERSHIP gives excessive privileges (D). Stored procedures with EXECUTE AS OWNER or 'EXECUTE AS CALLER', combined with limited schema privileges, offer the best control and security. 'EXECUTE AS OWNER' allows the procedure to run with the privileges of the app itself, bypassing consumer privileges, and 'EXECUTE AS CALLER' uses consumer's privileges, but still contains the data logic. Both combined, gives the best secure and recommended access control.

NEW QUESTION: 69

You are developing a Snowflake Native Application that provides data enrichment services. You want to offer tiered pricing based on the number of API calls made by consumers. Which of the following is the most effective approach to implement usage-based billing and track API call consumption within your application?

A. Implement a UDF within the application that increments a counter in a shared table each time an API endpoint is called. Query this table to determine usage.

B. Use Snowflake's resource monitors to limit the compute resources available to consumers based on their chosen tier.

C. Leverage Snowflake's built-in metering framework using and 'SYSTEM\$METER STAGE_STORAGE BYTES' to indirectly estimate API calls.

D. Implement a custom logging mechanism within the application to track API calls and store the logs in a separate Snowflake database, accessible only to the provider. Use scheduled tasks to analyze these logs and generate billing data.

E. Utilize Snowflake's events table to capture API call events based on function executions, enabling precise tracking and reporting of API usage for each consumer.

Answer: (SHOW ANSWER)

Option A is the most effective as it directly tracks API calls within the application. Resource monitors (B) only control compute costs, not API usage. System\$METER functions (C) track storage, not API calls. Custom logs (D) introduce complexity and security concerns. Snowflake's events table is a good general approach (E), but requires enabling detailed logging, which has performance implications and might not be available in all accounts. A shared table with a UDF provides the most direct and controllable way to meter usage.

NEW QUESTION: 70

A customer has installed your Snowflake Native Application, which utilizes a IJDF that performs complex calculations. The customer reports performance issues and sporadic errors. As the application provider, how can you BEST establish telemetry and observability to diagnose the problem, without compromising the customer's data security or privacy?

- A. Request direct access to the customer's Snowflake account to inspect the IJDF execution logs and query performance metrics.
- B. Implement logging within the UDF to capture execution time, input parameters (anonymized), and error messages. Store this data in an Event Table within the application provider's account and expose aggregate statistics through a secure API endpoint for the consumer to monitor.
- C. Utilize Snowflake's Query History to track the IJDF's performance in the customer's account and provide recommendations based on the observed query patterns.
- D. Instruct the customer to enable verbose logging in their Snowflake account and provide you with the log files for analysis.
- E. Create a shared view on the UDF execution logs in the customer's account, granting you read-only access to the relevant information.

Answer: ([SHOW ANSWER](#))

Option B provides a balanced approach to observability and security. By logging anonymized input parameters and execution time within the IJDF and storing the data in an Event Table in the application provider's account, you can analyze performance and identify errors without directly accessing the customer's sensitive data. Exposing aggregate statistics allows the customer to monitor the application's performance. The other options either compromise security or rely on the customer to provide data, which might not be feasible or secure.

NEW QUESTION: 71

You are designing a Snowflake Native Application that requires granular control over the privileges granted to different application roles. You want to follow the principle of least privilege. You have the following requirements: Application roles , and app_read_only'. What is the most effective strategy to implement this using definer's rights?

- A. Create stored procedures owned by the application, with each stored procedure explicitly granting the required privileges to the corresponding role. Consumers must then grant USAGE on these procedures to their desired roles.
- B. Create views for each application role, granting SELECT privilege on the necessary tables to each view. Then, grant SELECT privilege on the appropriate view to each application role.
- C. Grant all privileges on all application objects to the 'app_admin' role and rely on the consumer to manage privilege inheritance for other roles.
- D. Create separate schemas for each application role, placing objects with different privileges into these schemas. Then, grant the appropriate privileges on each schema to the corresponding application role.
- E. Create stored procedures owned by the application, with each stored procedure performing actions within the app. Grant OWNERSHIP on each stored procedure to the application admin. Consumers must then grant USAGE on these procedures to their desired roles.

Answer: ([SHOW ANSWER](#))

Using stored procedures owned by the application and using DEFINER's rights offers the most granular control and aligns with the least privilege principle. The application provider defines the privileges within the stored procedures. The consumer only needs to grant USAGE on the stored procedures. Creating views or separate schemas can work, but they add complexity. Granting all privileges to the 'app_admin' role is too permissive and doesn't adhere to the principle of least privilege.

NEW QUESTION: 72

You are developing a Snowflake Native Application and need to implement robust logging for troubleshooting purposes. You want to capture detailed information about function calls, including input parameters and return values. Which of the following approaches offers the MOST comprehensive and secure solution for logging within a Snowflake Native Application, considering the limitations imposed by the Snowflake environment?

- A. Using 'SYSTEM\$LOG' with trace level logging and store the logs into internal stage, accessible only by the application's service user.
- B. Employing 'EXECUTE IMMEDIATE' to write log messages directly to a dedicated logging table within the consumer's account.
- C. Leveraging Snowflake's event tables and configuring the application to emit custom events, then creating views on those event tables within the consumer's account.
- D. Writing log messages to an external stage (e.g., AWS S3, Azure Blob Storage) that is accessible to both the provider and consumer accounts.
- E. Using 'SYSTEM\$LOG' with debug level logging and capturing logs in a secure internal stage, managed by the provider, ensuring data privacy and limited consumer access.

Answer: ([SHOW ANSWER](#))

Option E is the most secure and comprehensive. 'SYSTEM\$LOG' allows for structured logging. An internal stage ensures that the consumer does not have direct access to sensitive log data. The provider maintains control, addressing security concerns. Options A and C are less secure, as they might expose sensitive application data. Option D is generally discouraged for sensitive information due to compliance concerns, and Option B is problematic because 'EXECUTE IMMEDIATE' inside an application package will be blocked.

NEW QUESTION: 73

You've identified a critical security vulnerability in your Snowflake Native Application that requires immediate remediation. You need to release a patch, but you want to minimize the impact on consumers who are already using your application. Which of the following approaches is MOST suitable for deploying this patch? Select TWO that apply.

- A. Create a new version of the application package with the patch applied, and immediately update the default version of the application package to the new version.
- B. Develop a SQL script containing the necessary fixes and provide it to consumers with instructions on how to apply it manually to their installed application instances.
- C. Create a new version of the application package containing the patch. Ensure the new version addresses the vulnerability while maintaining backward compatibility with existing application interfaces and data formats, and then gradually roll out the new version to consumers using APPLICATION roles and grants.
- D. Issue an official announcement to all consumers that they need to uninstall and reinstall the application to receive the security fix.
- E. Leverage zero-downtime patching (if supported by Snowflake in future) by updating internal objects without requiring consumers to update or redeploy their installed application instances.

Answer: ([SHOW ANSWER](#))

Options C and E are the most suitable. Option C prioritizes security while minimizing disruption by maintaining backward compatibility and rolling out the patch gradually. Consumers can update at their own pace without breaking existing workflows. Option E (if it were a feature) is optimal for security patching as it does not require any end-user action at all. Option A immediately updating default version could disrupt those not prepared for change. Option B places a burden on the consumer and opens the door for manual errors. Option D is disruptive and inconvenient for consumers.

NEW QUESTION: 74

A data provider wants to distribute a Snowflake Native Application that includes a secure view 'PROVIDER DB.PUBLIC.SALES SUMMARY' which aggregates sales data. They want to allow consumers of the application to query this view, but prevent them from accessing the underlying base tables. Which minimum set of privileges must be granted on the 'SALES SUMMARY' view to enable this, assuming the application role is 'APP ROLE' and the installing account doesn't want to use 'imported privileges'?

- A. GRANT SELECT ON VIEW PROVIDER DB.PUBLIC.SALES SUMMARY TO ROLE APP ROLE;
- B. GRANT USAGE ON DATABASE PROVIDER DB TO ROLE APP ROLE; GRANT USAGE ON SCHEMA PROVIDER DB.PUBLIC TO ROLE APP ROLE; GRANT SELECT ON VIEW PROVIDER DB.PUBLIC.SALES SUMMARY TO ROLE APP ROLE;
- C. GRANT ALL PRIVILEGES ON VIEW PROVIDER DB.PUBLIC.SALES SUMMARY TO ROLE APP ROLE;
- D. GRANT SELECT ON VIEW PROVIDER DB.PUBLIC.SALES SUMMARY TO SHARE WITH APP ROLE;
- E. GRANT REFERENCES ON VIEW PROVIDER DB.PUBLIC.SALES SUMMARY TO ROLE APP ROLE;

Answer: ([SHOW ANSWER](#))

The consumer needs 'USAGE' privilege on the database and schema containing the view to access it. The 'SELECT' privilege on the view itself is required to query it. 'GRANT ALL PRIVILEGES' is excessive. 'GRANT ... TO SHARE' is not a valid syntax. REFERENCES is for foreign keys.

NEW QUESTION: 75

You are developing a Snowflake Native Application intended for the Snowflake Marketplace. Before listing your application, you want to ensure it adheres to Snowflake's best practices for security, performance, and usability. Which of the following steps should you take to validate your application prior to submission? (Choose three)

- A. Manually review the application code to identify potential SQL injection vulnerabilities and optimize query performance.
- B. Use Snowflake's built-in code scanning tools (if available in the future) to automatically identify security vulnerabilities and code quality issues.
- C. Conduct thorough testing of the application on various Snowflake editions and sizes to ensure scalability and performance across different environments.
- D. Deploy the application to a separate Snowflake account and simulate user interactions to validate the application's usability and functionality.
- E. Submit your application code to a third-party security auditing firm for a comprehensive vulnerability assessment.

Answer: A,C,D ([LEAVE A REPLY](#))

Manual code review for SQL injection and performance (A) is crucial, as no automated tool currently perfectly covers all scenarios. Thorough testing (C) is essential to ensure scalability. Simulating user interactions in a separate account (D) helps validate usability and functionality. Third-party audits (E) are good practice but not explicitly required for initial Marketplace submission. Code scanning tools mentioned in (B) are a hypothetical scenario, which are not currently readily available on Snowflake marketplace at the time this response was constructed.

NEW QUESTION: 76

You are designing a Snowflake Native Application that needs to store temporary data for processing, which should be cleaned up when the application is uninstalled from the consumer account. Which of the following options would be the MOST appropriate and secure way to manage this temporary data?

- A. Create a temporary table in the application database using the APPLICATION role. The application is responsible for dropping the table when uninstalled.
- B. Create a table with a retention period in the application database using the APPLICATION role. Configure the retention period to be very short (e.g., 1 day).
- C. Create a schema named 'temp' in the application database using the APPLICATION role. The application setup script will create temporary tables inside this schema. Use an uninstall script to drop the entire schema upon uninstall.
- D. Create a temporary stage and use it to persist data. Use a SQL statement in your uninstall script to remove the data from the stage when the application is uninstalled.
- E. Create an external stage linked to cloud storage. Store temporary data in cloud storage and implement cleanup logic using an uninstall script.

Answer: ([SHOW ANSWER](#))

Creating a dedicated schema for temporary tables and dropping the entire schema in the uninstall script is the most secure and reliable way to manage temporary data. Using the APPLICATION role ensures that only the application can access and manage these tables. This approach guarantees that all temporary data is cleaned up when the application is uninstalled, without relying on external mechanisms or application logic running after uninstall.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 77

A software company, 'DataInsights', is developing a Snowflake Native Application that requires restricted access to a source data table in the consumer's account. DataInsights wants to ensure that only the application can access the data, and the consumer cannot directly query it. What is the MOST secure and recommended approach DataInsights should use within their application package to grant access?

- A. Granting the 'USAGE' privilege on the database and schema containing the source data table directly to the application role.
- B. Granting 'SELECT' privilege directly on the source data table to the application role.
- C. Creating a secure view on top of the source data table and granting 'SELECT' privilege to the application role on the view.
- D. Using a secure UDF within the application to access the source data table, ensuring data masking and redaction, with only the UDF being granted SELECT privileges on the table.
- E. Exposing the table directly through the application's setup script using 'GRANT SELECT ON TABLE TO APPLICATION ROLE app_role'.

Answer: ([SHOW ANSWER](#))

Using a secure UDF is the recommended approach. By encapsulating the data access within a secure UDF, you can control precisely how data is accessed and potentially mask sensitive information. Options A, B and E grants direct access which contradicts the requirement for data restriction. While a secure view (C) adds a layer of indirection, a UDF provides greater flexibility and control over the data access logic and potential redaction.

NEW QUESTION: 78

Consider the following Java UDF code snippet intended for use in a Snowflake Native Application. The UDF is designed to retrieve data from an external REST API. The code throws a 'java.net.SocketTimeoutException' intermittently when the API takes longer than expected to respond. How can you best handle this exception within the UDF to provide a more informative error message to the consumer and prevent the entire application from failing?

```
try {
    // Code to call REST API
} catch (java.net.SocketTimeoutException e) {
    throw new RuntimeException("API call timed out. Please try again later.");
}
```

A.

```
try {
    // Code to call REST API
} catch (java.net.SocketTimeoutException e) {
    System.err.println("API Timeout: " + e.getMessage());
    return null;
}
```

B.

```
try {
    // Code to call REST API
} catch (java.net.SocketTimeoutException e) {
    throw new SnowflakeSQLException("API_TIMEOUT", "API call timed out after multiple retries. Contact support if the issue persists.", e);
}
```

C.

```
try {
    // Code to call REST API
} catch (java.net.SocketTimeoutException e) {
    return "API Error: " + e.getMessage();
}
```

D.

```
try {
    // Code to call REST API
} catch (java.net.SocketTimeoutException e) {
    // Log the exception details to a separate logging service.
    return null;
}
```

E.

Answer: (SHOW ANSWER)

Option C is the best approach. Throwing a 'SnowflakeSQLException' allows you to provide a structured error message to the consumer with a custom error code (API_TIMEOUT) and a user-friendly message. It also includes the original exception for debugging purposes. Option A throws a generic 'RuntimeException' which is not specific to Snowflake and doesn't provide a clear error code. Option B only prints the error to standard error, which may not be captured or easily accessible within Snowflake. Option D returns a string which may not be the expected return type of the UDF and doesn't provide a structured error. Option E only logs the exception to an external service but the UDF needs to handle the error and return the correct snowflake error.

NEW QUESTION: 79

A Snowflake Native App provider wants to grant a consumer access to a secure view within their application. The provider's application package is named and the secure view is named 'sensitive_data_view'. The consumer's account identifier is 'XYZ12345'. Which of the following is the MOST secure and appropriate method to grant this access using Snowflake's Native App Framework?

- A. Grant SELECT on sensitive data view to account XYZ12345;
- B. Grant imported privileges on my_app_package to share with account XYZI 2345;
- C. Create a share and grant SELECT on sensitive_data_view to the share, then grant usage on the share to account XYZI 2345;
- D. Grant USAGE on database containing sensitive_data_view to account XYZI 2345;
- E. Create a role within the application package, grant SELECT on sensitive_data_view to the role, and create an authorization grant to account XYZ12345 for that role.

Answer: (SHOW ANSWER)

The most secure approach is to create a role within the application package specifically for accessing the secure view. By using an authorization grant, you explicitly control which consumer accounts can assume this role and access the view. Granting privileges directly to accounts or using shares is less secure and more difficult to manage in the context of a Native App. IMPORTED PRIVILEGES apply at the package level and are needed for a consumer to access the application, but aren't granular enough for specific objects.

NEW QUESTION: 80

A Snowflake Native App provider wants to implement row-level security (RLS) in an event table to ensure that consumers only receive events pertaining to their specific organization, even if they have been granted USAGE on the event table function. How can this be achieved?

- A. Create a masking policy that filters rows in the event table based on the CURRENT_ACCOUNT() function to match the consumer's account ID.
- B. Create a row access policy that filters rows in the event table based on the APP_INSTANCE_ID() function, ensuring each consumer only sees events related to their application instance.
- C. Implement a custom stored procedure that acts as a proxy for accessing the event table, filtering data based on the consumer's user ID.
- D. Create a view on top of the event table that includes a WHERE clause filtering by APP INSTANCE ID() and grant access to the view instead of the base table.
- E. Row-level security cannot be enforced on event tables shared through Snowflake Native Apps due to security limitations.

Answer: ([SHOW ANSWER](#))

B is the most appropriate answer. Row access policies can leverage APP_INSTANCE_ID() within the Snowflake Native App environment to filter data based on the application instance, achieving the desired row-level security. A is incorrect because CURRENT_ACCOUNT() would not distinguish between different application instances within the same account. C is incorrect as it is more complex than using row access policy directly and more difficult to maintain. D is similar to B but less secure than the row access policy. E is incorrect, RLS can be enforced.

NEW QUESTION: 81

Consider the following Snowflake Native Application design scenario: You are building a data enrichment application. This application requires a consumer to grant usage on a specific database in their account. The application will then use a user-defined function (UDF) to access and enrich the data. To minimize the risk of data exposure and adhere to best practices, which of the following approaches should you take when designing the UDF?

- A. Create the UDF with the 'VOLATILE' keyword to ensure it only runs when explicitly called, limiting the potential for unintended data access.
- B. Create the UDF with the 'SECURE' keyword and implement granular access control within the UDF code, restricting access based on the application role.
- C. Create the UDF with the 'IMMUTABLE' keyword to prevent any data modification within the consumer's database.
- D. Create the UDF without any special keywords, relying solely on the consumer's permissions to control data access.
- E. Create the UDF with the 'EXTERNAL_ACCESS' keyword to explicitly allow it to access data outside of the application package.

Answer: ([SHOW ANSWER](#))

The 'SECURE' keyword (Option B) is crucial for protecting data within a Snowflake Native Application. By creating a secure UDF, you can ensure that the UDF's code is protected from unauthorized inspection and modification. Furthermore, implementing granular access control within the UDF allows you to restrict data access based on the application's role, minimizing the risk of data exposure and adhering to best practices. 'VOLATILE', 'IMMUTABLE', and 'EXTERNAL_ACCESS' are irrelevant in this context. Not using any special keyword will not protect your data.

NEW QUESTION: 82

Consider a Snowflake Native App that dynamically creates tables within the consumer's account using stored procedures. The application role is 'APP_ROLE'. After installation, the application needs to automatically grant 'SELECT' privilege on all newly created tables in a specific schema (CAPP_DATA_SCHEMA) to a consumer-defined role (CONSUMER_ROLE). Which of the following approaches is the most secure and efficient way to achieve this? Assume the application cannot know the names of tables in advance.

- A. Use a task that runs periodically to discover new tables and grant SELECT privilege to CONSUMER_ROLE.
- B. Include code in the stored procedure that creates the table to immediately grant SELECT privilege to CONSUMER_ROLE.
- C. Grant OWNERSHIP on to the APP_ROLE with the GRANT OPTION, then have the application create the tables. The CONSUMER_ROLE will then have implicit access.
- D. Grant future grants on the schema for 'SELECT' privilege to the 'CONSUMER_ROLE' when tables are created.
- E. Manually create tables outside of the application and grant the privileges.

Answer: D ([LEAVE A REPLY](#))

Future grants are designed for this purpose: to automatically grant privileges on objects created in the future. Options A and B are less efficient and scalable. Option C is overly permissive, granting ownership and the ability to grant further privileges, which is a security risk. Manual creation defeats the purpose of automation.

NEW QUESTION: 83

You are troubleshooting a Snowflake Native Application installation that consistently fails with a cryptic error message: 'Internal Application Error: Check application logs for details'. You have access to the application's code package but are unsure how to effectively gather detailed logs and telemetry data from the running application. Which strategy would provide the MOST comprehensive and maintainable observability solution within the Snowflake Native App framework?

- A.** Implement calls within the application code to write log messages directly to a Snowflake table owned by the application provider. Consumers can then query this table to view logs.
- B.** Utilize the view to analyze query execution patterns and identify potential bottlenecks within the application's SQL code.
- C.** Integrate a third-party logging framework (e.g., Log4j) into the application code and configure it to write logs to an external logging service accessible over the internet.
- D.** Leverage Snowflake's event tables and configure the application to emit custom events to these tables. Consumers can then subscribe to these events to monitor application behavior.
- E.** Rely solely on the error messages returned by Snowflake during application execution, as these are sufficient for identifying most installation issues.

Answer: ([SHOW ANSWER](#))

Option A is the most comprehensive and maintainable solution. 'SYSTEM\$LOG' provides a native and secure mechanism for writing logs directly to a Snowflake table controlled by the application provider. This avoids the complexity and security risks of using external logging services (C) and provides structured, queryable log data within the Snowflake environment. B is helpful for performance, D is good, but not a native way. E relies on just error messages.

NEW QUESTION: 84

You are developing a Snowflake Native Application that leverages a UDF to perform complex data transformations. During provider-side testing in test mode, you encounter an intermittent error within the UDF. Which of the following strategies are MOST effective in debugging this issue within the limitations of test mode, considering you cannot directly access the consumer's Snowflake account or granular execution logs?

- A.** Utilize 'SYSTEM\$LOG' within the UDF to output detailed execution information to the application's event table. Analyze these logs after triggering the error in test mode.
- B.** Implement rigorous input validation within the UDF to catch potential data inconsistencies that might be causing the error. Ensure informative error messages are returned for debugging purposes.
- C.** Leverage Snowflake's query profiling tools on the provider side against the application's data-sharing objects to identify performance bottlenecks or unexpected behavior within the UDF's execution plan. (Assume data sharing objects are set up for provider side testing)
- D.** Simulate the consumer environment as closely as possible on the provider side by creating a separate Snowflake account with similar data volumes and configurations, then install the application in test mode in that account.
- E.** Use the ALERT' functionality to send notifications to the provider's account whenever the UDF encounters the error, allowing for real-time monitoring of the application's behavior.

Answer: ([SHOW ANSWER](#))

Options A and B are the most practical approaches within the constraints of test mode. allows you to capture execution information from the UDE Robust input validation (B) can help identify and isolate problematic data. Option C is less practical because you have limited access to query profiling tools. Option D is not ideal for debugging intermittently, while option E using alerts is not available in native apps context for UDF.

NEW QUESTION: 85

A Snowflake Native App developer wants to externalize certain configuration values so that a consumer can adjust the app's behavior without modifying the app's code directly. Which of the following methods is the MOST appropriate for achieving this using the Snowflake Native App Framework?

- A.** Hardcoding the configuration values directly into the SQL code.
- B.** Using Snowflake variables and setting them at the session level.
- C.** Defining parameters in the manifest file and using them within the application code.
- D.** Storing the configuration values in a separate Snowflake table within the consumer account.

E. Using dynamic SQL to modify the application code at runtime based on environment variables.

Answer: ([SHOW ANSWER](#))

Defining parameters in the manifest file is the most appropriate way to externalize configuration values in a Snowflake Native App. This allows the consumer to set these parameters during installation or upgrade, effectively tailoring the app's behavior without needing to modify the underlying code. Hardcoding values is inflexible, session variables are not persistent for the app, using a separate table adds unnecessary complexity, and dynamic SQL for code modification is generally discouraged for security and maintainability reasons.

NEW QUESTION: 86

You are designing a Snowflake Native Application package. You need to include a specific version of a Javascript UDF that calculates a complex statistical metric. You want to ensure that even if you update the application in the future, older installations continue to use the original Javascript UDF version. You also want to ensure that the javascript code is obfuscated. How do you design the application package directory structure and the UDF creation to achieve this? (Assume the UDF javascript source is named 'metric_v1.js' and is placed in the correct location within the application source code directory). The relevant 'manifest.yml' excerpt is:

```
- src: src/udfs
  versioned: true
```

- ☐ Place 'metric_v1.js' directly in the root of the application package. Create the UDF using `CREATE OR REPLACE FUNCTION calculate_metric() RETURNS FLOAT LANGUAGE JAVASCRIPT AS $$ ...content of metric_v1.js... $$;`
- ☐ Create a subdirectory 'udfs' in the application package, place 'metric_v1.js' within it. Create the UDF using `CREATE OR REPLACE FUNCTION calculate_metric() RETURNS FLOAT LANGUAGE JAVASCRIPT AS $$ ...content of metric_v1.js... $$;`
- ☐ Create a subdirectory 'udfs' in the application package, place 'metric_v1.js' within it. Run the `snowflake-cli upload` command to obfuscate the file to 'metric_v1_obfuscated.js'. Create the UDF using `CREATE OR REPLACE FUNCTION calculate_metric() RETURNS FLOAT LANGUAGE JAVASCRIPT AS $$ ...content of metric_v1_obfuscated.js... $$;`
- ☐ Create a subdirectory 'udfs' in the application package, place 'metric_v1.js' within it. Run the `snowflake-cli upload` command to obfuscate the file to 'metric_v1_obfuscated.js'. Create the UDF using `CREATE OR REPLACE FUNCTION calculate_metric() RETURNS FLOAT LANGUAGE JAVASCRIPT AS $$ ...content of @udfs/metric_v1_obfuscated.js $$;`
- ☐ Create a subdirectory 'udfs' in the application package, place 'metric_v1.js' within it. Run the `snowflake-cli upload` command to obfuscate the file to 'metric_v1_obfuscated.js'. Create the UDF using `CREATE OR REPLACE FUNCTION calculate_metric() RETURNS FLOAT LANGUAGE JAVASCRIPT AS $$ ...content of @udfs/metric_v1.js $$;`

- A. Option A
- B. Option B
- C. Option C
- D. Option D
- E. Option E

Answer: ([SHOW ANSWER](#))

Option E correctly addresses the requirements. By creating the 'udfs' subdirectory and placing 'metric_v1.js' within it, and 'versioned: true' inside the 'manifest.yml' file allows for versioning of the script. Use of the .js' syntax within the UDF definition ensures that the correct version is referenced even after application updates. By using the upload command, this ensures that the file is obfuscated, and by referencing the original name in the UDF creation, this will ensure that the obfuscated version is used for all consumer instances. Options A and B do not provide versioning or obfuscation. Option C is incorrect because it does not utilize Snowflake's versioning or correct syntax, and the uploaded, obfuscated javascript can no longer be referenced. Option D is incorrect since the command upload command does not rename the js file, and the file would not be referenced correctly.

NEW QUESTION: 87

An application provider wants to grant usage privileges on a warehouse named 'APP WAREHOUSE' to a consumer account so their application can use the warehouse's compute resources. Which of the following SQL statements would correctly accomplish this goal, ensuring only the necessary privileges are granted to the consumer's application role 'APP ROLE'?

- A. `GRANT USAGE ON WAREHOUSE APP_WAREHOUSE TO APPLICATION ROLE APP_ROLE;`
- B. `GRANT ALL PRIVILEGES ON WAREHOUSE APP_WAREHOUSE TO APPLICATION ROLE APP_ROLE;`
- C. `GRANT USAGE ON WAREHOUSE APP_WAREHOUSE TO ROLE APP_ROLE;`
- D. `GRANT OPERATE ON WAREHOUSE APP_WAREHOUSE TO APPLICATION ROLE APP_ROLE;`
- E. `GRANT MONITOR ON WAREHOUSE APP_WAREHOUSE TO APPLICATION ROLE APP_ROLE;`

Answer: ([SHOW ANSWER](#))

The correct statement is 'GRANT USAGE ON WAREHOUSE APP_WAREHOUSE TO APPLICATION ROLE APP_ROLE;'. This grants the application role the necessary privilege to use the warehouse for compute. 'GRANT ALL PRIVILEGES' is too broad and grants unnecessary permissions. Granting to a ROLE' (Option C) instead of an 'APPLICATION ROLE' is incorrect for granting privileges to a Native Application. 'OPERATE' and 'MONITOR' are not valid privileges for warehouses in this context; only 'USAGE' is typically required for an application to utilize a warehouse.

NEW QUESTION: 88

You are developing a Snowflake Native Application. During internal testing, you need to simulate different consumer account environments to validate the application's behavior under various conditions (e.g., different Snowflake editions, security configurations). What are the recommended methods for setting up these test environments WITHIN the constraints of the Snowflake Native App framework?

- A. Create multiple Snowflake accounts, each representing a different consumer environment, and install the application in each account.
- B. Use Snowflake's cloning capabilities to create multiple clones of your developer account, each with different configurations, and install the application in each clone.
- C. Develop a configuration file within the application that allows you to simulate different consumer environments. Implement logic in the application code to adapt to these simulated environments.
- D. Utilize Snowflake's Resource Monitors to limit the resources available to the application, simulating a consumer environment with limited compute.
- E. Employ multiple developer accounts with different configurations to simulate consumer environments, package your application from each account and test it. Additionally, use parameters within the setup script to configure application behavior.

Answer: C,E ([LEAVE A REPLY](#))

While creating multiple Snowflake accounts (Option A) is a valid approach, it can be resource-intensive. Option C & E (Employ configuration files within the application and Multiple Developer Accounts) is the MOST practical within the Snowflake Native App framework. By developing configuration files and multiple developer accounts and their setups, the application can adapt its behavior to simulate different consumer environments without the overhead of managing multiple full Snowflake accounts. Using parameters during installation also adds flexibility.

NEW QUESTION: 89

You are tasked with deploying a new version of your Snowflake Native Application. The new version introduces significant schema changes to the data warehouse used by the application. What steps should you take during the deployment process to ensure a smooth transition and minimize downtime?

- A. Immediately replace the existing application version with the new version. Snowflake will automatically handle any schema compatibility issues.
- B. Deploy the new version as a staging version alongside the existing version. Implement data migration scripts to gradually move data to the new schema, and then switch the production traffic to the new version.
- C. Notify all users that the application will be unavailable for several hours while the schema changes are applied directly to the production environment.
- D. Use Snowflake's zero-copy cloning feature to create a copy of the existing database. Apply the schema changes to the cloned database, test thoroughly, and then promote the cloned database to production.
- E. Modify the existing tables in place to reflect the new schema. Update your application's code to be compatible with the new schema after the changes are complete.

Answer: ([SHOW ANSWER](#))

Options B and D are both valid approaches for deploying a new version with schema changes. Option B provides a controlled and gradual migration to the new schema. Option D leverages Snowflake's cloning capabilities to minimize downtime and provide a rollback option. Option A assumes automatic handling of schema changes, which is not correct. Option C results in significant downtime. Option E is risky and can lead to data inconsistencies and application errors if not handled carefully and incrementally. Promoting the clone to production is preferable to modifying in place.

NEW QUESTION: 90

You are developing a Snowflake Native Application that uses Snowpark Container Services. Your application consists of multiple containerized services that require varying levels of computational resources. You want to optimize resource utilization and minimize costs by using a single Compute Pool for all services, while ensuring that each service has guaranteed access to resources. Which of the following strategies and SQL commands (or configurations) can be used to achieve this?

- A. Create separate Compute Pools for each service, sizing each pool according to the service's peak resource requirements.
- B. Use a single Compute Pool and rely on the container orchestration system to dynamically allocate resources among the services. Monitor utilization and adjust the Compute Pool size as needed. No SQL commands are needed.
- C. Create a single Compute Pool and configure resource quotas for each service using container orchestration-specific configurations (e.g., Kubernetes resource quotas). While not directly managed via SQL, proper configuration of the container orchestrator is required.
- D. Use a single Compute Pool, create separate user-defined functions (UDFs) for each service and control resource allocation using the 'RESOURCE MONITOR parameter for each UDF.
- E. Create a resource monitor and assign it to the COMPUTE POOL to manage cost. Resource monitors cannot be used to granular control the compute pool usage for native applications.

Answer: B,C (LEAVE A REPLY)

Option B is valid because the underlying container orchestration system dynamically allocates resources. Option C is also valid because resources can be configured in container orchestration systems to achieve resource guarantees for each service. Creating separate compute pools may be costly and inefficient in terms of resource management. Resource Monitors are for cost control not fine-grained service-level resource management. It provides control to monitor cost. The correct configuration of the container orchestrator is critical to achieve desired outcomes.

NEW QUESTION: 91

You are developing a Snowflake Native Application that relies on external functions (UDFs or external functions using API integration) to enrich data. After installing the application in a customer's account, you observe that the external functions are failing intermittently. The error messages indicate network connectivity issues. Which of the following troubleshooting steps would be MOST effective in diagnosing and resolving this problem from the provider's perspective, assuming you cannot directly access the customer's network configuration?

- A. Instruct the customer to check their network firewall rules to ensure that outbound traffic to your API endpoint is allowed. Provide them with a list of IP addresses for your service.
- B. Implement retry logic within the external function calls with exponential backoff. Log all failed attempts, including timestamps and error messages, within your provider's environment, for later analysis. Provide the customer with aggregated success/failure metrics from your logs.
- C. Create a separate Snowflake Native App that only tests network connectivity to the external API endpoint. Ask the consumer to install and run the application, and provide you with the results.
- D. Migrate the external functions to use Snowflake's Snowpark Container Services to ensure they run within the Snowflake environment and bypass any network restrictions.
- E. Monitor the external functions calls from system \$ table and analyze the latency and error codes.

Answer: (SHOW ANSWER)

Option B is the most effective approach because it combines resilience (retry logic) with observability (logging). By implementing retry logic with exponential backoff, the application can automatically recover from transient network issues. Logging failed attempts within the provider's environment allows you to analyze the problem without needing direct access to the customer's network configuration. Providing aggregated metrics to the customer allows them to monitor the application's health and identify potential network issues on their end. Option A requires the customer to have technical expertise and exposes your infrastructure details. Options C can add overhead for the consumer. Option D is costly, time consuming, or inappropriate and Option E needs external configuration.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 92

You are developing a Snowflake Native Application and want to implement robust observability and telemetry. Which of the following approaches will enable you to effectively monitor the application's performance, identify errors, and collect usage metrics within the consumer's account?

- A.** Rely solely on Snowflake's built-in query history and resource monitoring features in the consumer's account, as these automatically capture all application activity.
- B.** Implement custom logging using 'SYSTEM\$LOG' and store application logs in a separate table within the application's container. Configure grants to allow the application developer to access these logs.
- C.** Utilize to write log data to a secure table within the application and leverage views granted to the consumer to expose relevant metrics and error information. Ensure appropriate roles have access to the views.
- D.** Directly access the consumer's system tables (e.g., 'SNOWFLAKE.ACCOUNT USAGE.QUERY HISTORY') to extract application-related metrics based on query identifiers generated by the application.
- E.** The consumer is responsible for implementing observability and telemetry. The application developer cannot implement observability features that function within the consumer's account.

Answer: (SHOW ANSWER)

Option C is the most effective approach. Using 'SYSTEM\$LOG' allows for structured logging within the application. Storing the logs in a secure table within the application container keeps the data isolated and controlled. Creating views and granting access to specific roles on the consumer side allows the application to expose relevant metrics and error information without granting direct access to the underlying log data. Options A, D, and E are incorrect because they do not provide a comprehensive and controlled observability solution managed by the application developer. Option B is incorrect because the application developer cannot directly access data created in the consumer's account after installation.

NEW QUESTION: 93

A provider has released a Snowflake Native Application that calculates risk scores. As a consumer, you've installed the application. However, you are seeing inconsistent results when querying a view provided by the application named 'RISK SCORES'. You suspect the provider has an issue with their data or logic. Which of the following actions are MOST appropriate to troubleshoot this from the consumer side, focusing on querying and without access to the provider's source code?

- A.** Contact Snowflake Support and provide them with a detailed description of the issue and relevant query examples. Snowflake Support can access the provider's environment and investigate.
- B.** Use on the provider's account to establish a secure connection and directly debug the provider's stored procedures.
- C.** Examine the 'INSTALLATION INFORMATION' view in the application's database to check for provider-published error messages or status updates related to data quality issues. Use queries to identify patterns in the inconsistent results.
- D.** Utilize the 'DEBUG' procedure exposed by the application (if any) to step through the risk calculation logic with specific input values.
- E.** Raise a support ticket with the provider through the Snowflake Marketplace. Include example queries, input data, and the expected vs. actual results to facilitate their debugging process. Further, Query data using different warehouse sizes to see if concurrency is a factor.

Answer: (SHOW ANSWER)

Options C and E are correct. As a consumer, you primarily interact with the application through its exposed interface (views, procedures). Option C allows you to check for provider-published information about issues. Option E is the standard way to report issues to the provider. Option A is incorrect because Snowflake Support generally doesn't have access to provider environments in this scenario. Option B is incorrect because consumers do not have direct access to provider accounts. Option D depends on the application's features; if exposed by the application it can be used but not all native apps do. Finally, the warehouse size in the consumer account can sometimes influence the behavior of the consumer side query against the views.

NEW QUESTION: 94

Consider the following Python code snippet used within a Streamlit application deployed as a Snowflake Native App:

- A.** The code will execute without any privilege errors, assuming the necessary database and schema privileges are granted to the 'app_public' role.
- B.** The code will fail because the Snowflake Connector for Python requires explicit connection parameters to be defined even within a Native App context.

- C. The code will fail with a privilege error if the 'app_public' role does not have OWNERSHIP of the table 'my_table' .
- D. The code will execute successfully, assuming that 'my_table' exists, and the 'app_public' role has at least SELECT privilege on 'my_table' and USAGE privilege on the schema containing 'my_table' .
- E. The code requires using the 'snowflake.snowpark' library instead of for optimized performance within native apps.

Answer: ([SHOW ANSWER](#))

The correct answer is D. When running within a Snowflake Native App, the Snowflake Connector for Python automatically uses the current session context, eliminating the need for explicit connection parameters. The code will execute successfully provided that the 'app_public' role has the appropriate privileges, which includes 'SELECT on 'my_table' and 'USAGE on the schema containing the table. Option A is partially correct, but option D provides more specific privilege requirements. Option B is incorrect because explicit connections aren't needed. Option C is incorrect because 'OWNERSHIP' isn't required, only 'SELECT. Option E is incorrect; using 'snowflake.snowpark' may offer performance benefits for data processing, but using "snowflake.connectofl is viable for simple queries.

NEW QUESTION: 95

A data analytics company, 'Data Insights Pro', is developing a Snowflake Native Application to provide advanced customer segmentation analysis. They plan to monetize this application through the Snowflake Marketplace. They want to ensure that only a specific set of customer accounts can access and install their application during the initial beta testing phase. Which of the following steps must they perform to achieve this?

- A. Create a public listing on the Snowflake Marketplace and rely on Snowflake's built-in usage monitoring to identify the intended customer accounts.
- B. Create a private listing on the Snowflake Marketplace and share it with the specific customer accounts by explicitly granting access to their Snowflake account identifiers.
- C. Publish the application directly to the Snowflake Marketplace without creating a listing, and then share the installation URL with the selected customer accounts.
- D. Use Snowflake's data sharing feature to share the application package with the customer accounts.
- E. Create a public listing with a very high price to deter unintended installations, then manually adjust the price to zero for beta testers.

Answer: ([SHOW ANSWER](#))

A private listing allows you to control exactly which Snowflake accounts can see and install your application. Public listings are visible to everyone, and other options don't provide the necessary control over access during beta testing. Data sharing isn't used for native app distribution.

NEW QUESTION: 96

You are developing a Snowflake Native App that requires specific database privileges to be granted to the application role during installation. Which sections of the 'manifest.yml' file are primarily responsible for defining these required privileges, and how does Snowflake interpret and enforce them during the installation process?

- A. The 'application.initial_version.privileges' section defines the privileges needed. Snowflake automatically grants these privileges to the application role defined in 'application.initial_version.role' before the setup script runs. This ensures the application has the necessary permissions to create objects and execute code within the consumer's account.
- B. The 'setup.script' section defines the privileges needed. The application setup code must explicitly grant these privileges using 'GRANT statements to a specific application role. Snowflake does not automatically grant any privileges, so you have full control over permission management.
- C. The 'application.initial_version.artifacts' section is used to define the privileges. Snowflake parses this section to identify required privileges and grants them dynamically to the application role whenever a function or procedure is called that requires them.
- D. A combination of the 'application.initial_version.privileges' and 'setup.script' sections. 'application.initial_version.privileges' defines initial static privileges, while 'setup.script' can grant additional privileges dynamically during installation if needed. Snowflake ensures that all specified privileges are granted before the application is fully operational.
- E. The 'application.initial_version.parameters' section. While parameters can control behavior, they do not define or manage privileges. Privileges must be manually configured by the consumer post-installation.

Answer: ([SHOW ANSWER](#))

The correct answer is D. The 'application.initial_version.privileges' section is used to define static privileges that the application role should have from the start. The 'setup.script' allows for more dynamic privilege granting, enabling the application to adjust permissions based on the consumer's environment or configuration. Snowflake ensures these privileges are in place to enable the application to function correctly.

NEW QUESTION: 97

You are developing a Snowflake Native Application that includes a TASK which calls a stored procedure. The stored procedure interacts with an external API using an external function. The external function is defined with an API integration. To ensure secure access to the external API, how should the privileges be granted so that the TASK can execute successfully and access the external function within the consumer account?

- A.** Grant USAGE privilege on the API integration and EXECUTE TASK privilege on the TASK to the 'app_public' role.
- B.** Grant OWNERSHIP on the API integration and EXECUTE TASK privilege on the TASK to the 'app_public' role.
- C.** Grant USAGE privilege on the API integration to the 'app_public' role and no specific privilege on the TASK, as task execution is implicitly granted.
- D.** Grant USAGE privilege on the API integration and EXECUTE TASK privilege on the TASK to the 'app_owner' role.
- E.** Grant USAGE privilege on the API integration to the 'application' role and EXECUTE TASK privilege on the TASK to the 'app_public' role.

Answer: ([SHOW ANSWER](#))

For the TASK to successfully call the stored procedure which in turn calls the external function using API integration, the 'app_public' role needs 'USAGE' privilege on the API integration and 'EXECUTE TASK' privilege on the TASK. The 'app_owner' role is typically used for managing the application and not for executing tasks in the consumer account. The 'application' role doesn't exist in the context of native apps in this sense. Granting USAGE on API integration allows application to use the integration. Granting execute on task, enables the task to be executed.

NEW QUESTION: 98

A Snowflake Native App developer is creating an application that needs to access data from multiple tables within a provider account. The developer wants to ensure the principle of least privilege and minimize the surface area for potential security vulnerabilities. Which approach is the MOST secure and efficient for granting the necessary privileges to the application?

- A.** Grant the 'OWNERSHIP' privilege on all relevant databases to the application role. This provides the application with full control over the data, ensuring it can perform any necessary operations.
- B.** Grant the USAGE privilege on all relevant databases and schemas, and the 'SELECT' privilege on only the tables required by the application to the application role.
- C.** Grant the 'ALL PRIVILEGES' privilege on all relevant tables to the application role. This provides the application with all possible privileges, simplifying privilege management.
- D.** Create a custom role in the provider account, grant the required privileges on the tables to this role, and then grant the 'USAGE' privilege on this custom role to the application role.
- E.** Grant the 'SELECT' privilege on all tables in the provider account to the application role.

Answer: ([SHOW ANSWER](#))

Option B is the MOST secure and efficient. Granting 'USAGE' on databases and schemas allows the application to access these containers, while granting 'SELECT' only on the necessary tables adheres to the principle of least privilege, minimizing potential security risks. Options A, C, and E grant overly broad privileges, which is not a secure practice. Option D, while seemingly applying least privilege, creates unnecessary role complexity and doesn't directly link the application role to the required object privileges.

NEW QUESTION: 99

You are developing a Snowflake Native Application that utilizes Snowpark Container Services to perform complex image processing. The application needs to securely access data stored in the consumer's Snowflake account, specifically a table named 'USER IMAGES' in the schema 'RAW DATA'. The container image is built and pushed to Snowflake's internal registry. Which of the following steps are necessary to ensure the container service can access this data securely and efficiently within the consumer's account without exposing credentials directly in the container environment?

- A.** Grant the SNOWFLAKE CONTAINER SERVICE role to the account administrator and use the administrator's credentials within the container service's code.
- B.** Create a Snowflake service account and grant it SELECT privileges on the 'RAW DATA.USER IMAGES' table. Store the service account's credentials as environment variables within the container service's specification.
- C.** Create a Snowflake network rule that allows access to all Snowflake internal IPs. Associate this rule with the container service's network policy.
- D.** Create a Snowflake database role, grant SELECT privileges on ' to this role, and then assign this role to the service account used by the container service through an entitlement.
- E.** Configure a Snowflake OAuth integration and associate it with the container service. The container application would then use the OAuth client credentials to obtain an access token and authenticate with Snowflake.

Answer: D ([LEAVE A REPLY](#))

The most secure and efficient way to grant access is to use database roles and entitlements. Creating a database role with the required privileges, assigning it to the service account of the container service through an entitlement provided to consumer is the recommended approach. This avoids storing credentials directly and provides fine-grained access control. Using SNOWFLAKE_CONTAINER_SERVICE

role is too broad and admin credentials should never be used directly in the code. Storing service account credentials in environment variables is insecure. OAuth is not directly used for container services accessing data within the consumer's account in this manner; entitlements offer a more streamlined and secure approach. Network Rules control egress traffic, not access to Snowflake data within the account.

NEW QUESTION: 100

You are developing a Snowflake Native App that processes sensitive customer data. To comply with regulations, you need to ensure that data remains within the customer's Snowflake account and is not exposed to your provider account. Which of the following architectural approaches are MOST effective for achieving this goal?

- A. Utilize external functions with secure data sharing, ensuring all processing logic resides in the consumer's account.
- B. Implement data masking policies within the provider account and share only the masked data with the consumer.
- C. Use secure UDFs deployed within the consumer's account, allowing the app to process data directly without moving it.
- D. Leverage external stages to store data outside of Snowflake, and use the app to access the data through the stage connection.
- E. Employ data clean room functionality, leveraging agreed-upon procedures and technical controls to prevent inappropriate data disclosure.

Answer: ([SHOW ANSWER](#))

Options A, C, and E directly address the requirement of keeping data within the customer's account. External functions (with secure data sharing) and secure UDFs allow processing within the consumer's environment. Data clean rooms provide a secure environment for collaboration and data analysis while minimizing data transfer. Option B violates the requirement by processing data in provider account. External stages (D) do not guarantee data residing within consumer's snowflake environment.

NEW QUESTION: 101

You are developing a Snowflake Native Application that allows customers to enrich their data. The application involves executing Javascript UDFs. Which of the following practices are considered unacceptable regarding code practices within the UDFs to ensure the Native App complies with Snowflake's requirements and security best practices?

- A. Using the SYSTEM\$ALLOWLIST function to restrict external network access from the Javascript UDFs to a specific set of approved domains. Using Snowflake's network rules is also acceptable for newer accounts.
- B. Directly embedding secrets (e.g., API keys, passwords) within the Javascript UDF code itself, even if obfuscated.
- C. Using JavaScript's eval() function to dynamically execute code based on user input or data retrieved from external sources.
- D. Relying solely on the consumer's network policy to manage external network access from the UDFs.
- E. Using Javascript UDFs to access data not needed by the application to save on querying costs.

Answer: ([SHOW ANSWER](#))

Options B, C, D and E are all unacceptable. Embedding secrets (B) directly in code is a major security risk. Using 'eval()' (C) opens the door to code injection vulnerabilities. Relying solely on the consumer's network policy (D) is insufficient; the application provider must also implement security measures. Javascript UDFs should only access required data by the application as this is also one of best practices regarding code practices within the UDFs. Option A is acceptable as it implements whitelisting of external network access.

NEW QUESTION: 102

A Snowflake Native App aims to provide data masking as a service. The app needs to access customer data, apply masking policies based on app-managed configuration, and store the masked data either in the customer's account or in the app's managed account based on customer preference. Which of the following steps and considerations are CRUCIAL for designing this application, focusing on security and privilege management? (Select all that apply)

- A. Request the OWNERSHIP privilege on the customer's tables to ensure full control over data masking.
- B. Implement the data masking logic within stored procedures that execute with the CALLERS RIGHTS to leverage the customer's existing masking policies.
- C. Use secure views to access the customer's data, ensuring that the app only sees the data it is authorized to access based on requested READ privileges.
- D. Ensure the application uses the SNOWFLAKE.CORE.INVOKE FUNCTION to call data masking policies with the consumer's data.
- E. Use stored procedures that execute with DEFINER'S RIGHTS and request necessary READ privileges on the customer's database and tables.

Answer: C,E ([LEAVE A REPLY](#))

The correct answers are C and E. Requesting OWNERSHIP (A) is excessive and poses a security risk. Using CALLER'S RIGHTS (B) would mean the app's functionality relies on the installing user's privileges and won't work if they don't have sufficient access. The INVOKE_FUNCTION is not needed. Using DEFINER'S RIGHTS (E) allows the app to execute with its own defined privileges, ensuring consistent behavior regardless of the installing user. Secure views (C) limit the app's access to only the authorized data, enhancing security.

NEW QUESTION: 103

You're developing a Snowflake Native Application that requires frequent updates to its core logic. You need to implement a deployment strategy that minimizes downtime and allows for easy rollback in case of issues. Which combination of deployment techniques would BEST achieve this goal when deploying updates to your Snowflake Native App? (Choose two)

- A.** Utilize zero-copy cloning to create a staging environment of your application, deploy the new version to the clone, and then swap the clone with the production environment for near-instant deployment.
- B.** Deploy the updated application directly to the production environment during off-peak hours, relying on thorough testing in a pre-production environment to minimize risks.
- C.** Implement a blue/green deployment strategy, maintaining two identical environments. Deploy the update to the 'blue' environment, test, and then switch traffic from 'green' to 'blue'. If issues arise, switch back to 'green'.
- D.** Use the ALTER APPLICATION' command with the 'FORCE' option to immediately replace the existing application with the new version, ignoring any potential compatibility issues.
- E.** Implement canary deployments by gradually rolling out the new version to a small subset of consumers, monitoring performance, and then expanding the rollout if no issues are detected.

Answer: ([SHOW ANSWER](#))

The best options are A and E. Zero-copy cloning allows for rapid deployment and rollback, as the switch between environments is metadata-only. Canary deployments allow for testing in production with minimal impact on the majority of users. Option C, while a valid strategy in general, isn't directly applicable in Snowflake Native Apps where you are providing an application into a customer's account. Options B and D are risky, as they involve directly modifying the production environment without sufficient safeguards. Force option is risky and dangerous.

NEW QUESTION: 104

You are developing a Snowflake Native Application and wish to publish an application package to an external staging location (e.g., AWS S3). You have already created the application package in Snowflake. What are the necessary steps and considerations to ensure the application package is securely and correctly deployed to the external stage, making it accessible to consumers?

- A.** Grant the SNOWFLAKE DEPLOYMENT application role the CREATE STAGE privilege on the external stage. Then, use the CREATE APPLICATION PACKAGE command with the EXTERNAL STAGE parameter pointing to the external stage location.
- B.** Create an IAM role in AWS with appropriate permissions to access the S3 bucket. Create a Snowflake integration object referencing the IAM role. Use the CREATE APPLICATION PACKAGE command with the EXTERNAL STAGE parameter pointing to the Snowflake integration object.
- C.** Use the PUT command to directly upload the application package files to the external stage. Then, create an APPLICATION using the CREATE APPLICATION command, referencing the uploaded files in the stage using the EXTERNAL_STAGE parameter.
- D.** Grant the ACCOUNTADMIN role to the SNOWFLAKE DEPLOYMENT application role and use the TOKEN function to create credentials. Use these credentials to upload the application package files to the external stage.
- E.** Create an APPLICATION PACKAGE, then grant CREATE APPLICATION privilege on the package to SNOWFLAKE DEPLOYMENT role and use the SYSTEM\$GENERATE_SCIM_ACCESS_TOKEN function to create credentials. Use these credentials to upload the application package files to the external stage.

Answer: ([SHOW ANSWER](#))

To securely and correctly deploy an application package to an external stage, you need to create an IAM role in AWS (or equivalent in other cloud providers) with the necessary permissions to access the S3 bucket (or equivalent). Then, create a Snowflake integration object that references this IAM role. Finally, when creating the APPLICATION PACKAGE, the EXTERNAL STAGE parameter should point to this Snowflake integration object. This ensures Snowflake can securely access the external stage. Option A is incorrect because it directly grants CREATE STAGE to the SNOWFLAKE DEPLOYMENT role on the external stage, bypassing proper authentication and authorization. Options C and D do not correctly leverage Snowflake's security mechanisms for external stage access. E grant create applicaiton privilege which is not appropriate for package creation.

NEW QUESTION: 105

A provider is developing a Snowflake Native App and utilizes event sharing to communicate state changes to consumers. After initial installation, a consumer reports not receiving any event notifications. The provider has verified the event table is populated correctly. Which of the following are the MOST likely reasons for this issue and how can the provider troubleshoot them?

- A.** The consumer's event table function does not have the correct security invoker privileges. Grant EXECUTE TASK on the event table function to the consumer role.
- B.** The consumer has not explicitly granted the necessary privileges on the event table function to the role used by the app. Instruct the consumer to grant USAGE on the event table function to the app role.
- C.** The consumer has not refreshed their application instance metadata. Ask the consumer to execute SYSTEM\$REFRESH_EXTERNAL_FUNCTIONS() in the application database.
- D.** The event table function is not correctly filtering events based on the consumer's application instance ID. Review the function logic to ensure proper filtering using APP_INSTANCE_ID().
- E.** Event sharing is inherently unreliable and Snowflake does not guarantee event delivery. The provider should implement alternative polling mechanisms.

Answer: ([SHOW ANSWER](#))

B and D are the most likely reasons. B: The consumer needs to grant USAGE on the event table function to the application role. D: The event table function must correctly filter events using APP_INSTANCE_ID() to ensure the consumer only receives events related to their instance. A is incorrect because the consumer (not provider) needs to grant usage. C is incorrect as SYSTEM\$REFRESH_EXTERNAL_FUNCTIONS() is not the correct procedure in this context. E is wrong. Snowflake guarantees event delivery, assuming correct setup.

NEW QUESTION: 106

A Snowflake Native App provider has implemented an Event Table to capture application logs. They observe a significant performance degradation in their application queries when the Event Table grows beyond a certain size. Which of the following strategies can be implemented to mitigate the performance issue, assuming that frequent querying of the last 7 days of data is common? Select ONLY ONE answer.

- A.** Increase the warehouse size used for querying the Event Table.
- B.** Create a materialized view on top of the Event Table that filters for events within the last 7 days.
- C.** Partition the Event Table by date, using the 'INSTALLATION_TIMESTAMP' column.
- D.** Implement a stored procedure that manually copies the last 7 days of data to a separate, smaller table.
- E.** Disable automatic clustering on the Event Table.

Answer: ([SHOW ANSWER](#))

Option B, creating a materialized view that filters for the last 7 days of data, is the MOST effective solution. Materialized views pre-compute and store the results of the query, so subsequent queries against the view are significantly faster. Increasing the warehouse size (Option A) might provide some improvement but does not address the fundamental issue of scanning the entire table. Partitioning (Option C) is not supported in Snowflake. Implementing a stored procedure (Option D) adds complexity and overhead. Disabling automatic clustering (Option E) will degrade performance.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 107

A consumer installs a Snowflake Native Application from the Marketplace. The application requires access to a specific table, 'CUSTOMER DATA', in the consumer's account. Which of the following steps are essential for the consumer to properly grant the application the necessary permissions, following least privilege principles, and ensuring the application functions as intended? Assume the application developer has properly requested the table grant in the setup script.

- A.** Grant the 'IMPORTED PRIVILEGES' privilege on the database containing 'CUSTOMER_DATA' to the application role created during installation.
- B.** Grant the 'SELECT' privilege directly on the 'CUSTOMER_DATA' table to the 'SNOWFLAKE.APP_ROLE' role associated with the application instance.
- C.** Create a share from the provider account granting access to the 'CUSTOMER_DATA' table and import it into the consumer account.
- D.** Grant the 'USAGE' privilege on the database and schema containing 'CUSTOMER_DATA' to the application role and then grant 'SELECT' on the 'CUSTOMER_DATA' table to the application role.
- E.** Grant 'OWNERSHIP' on the 'CUSTOMER_DATA' table to the application role. This provides the application with complete control over the table.

Answer: (SHOW ANSWER)

Option D is correct because it grants the minimum required privileges for the application to access the 'CUSTOMER_DATX' table. Granting 'USAGE' on the database and schema allows the application to navigate to the table. Granting 'SELECT' on the table itself allows the application to read the data. Options A, B, C and E are incorrect because they either grant excessive privileges, are not the correct approach for Native Apps (Shares), or grant permissions to the wrong role.

NEW QUESTION: 108

You are developing a Snowflake Native Application that connects to an external API to enrich data. To comply with best practices and maintain security posture of your Native App, you decide to manage the API Key as Secret. Which of the following approaches are needed to ensure that the Snowflake Native application can securely access the secret?

- A. Store the API Key as a Secret within Snowflake and grant USAGE privilege on the secret to the application role.
- B. Store the API Key directly within the application code, encrypting it using AES encryption with a key managed by Snowflake.
- C. Create a secure UDF that retrieves the API Key from an external key management system and grant USAGE privilege on UDF to the application role.
- D. Store the API Key as an environment variable within the application container.
- E. Store the API key using Snowflake's secret management feature and the 'USAGE' privilege to the application's invoker's rights role.

Answer: A,E (LEAVE A REPLY)

Snowflake's secret management feature with appropriate privilege grants is the recommended approach. Storing the API Key as a Secret (A & E) within Snowflake and granting 'USAGE' privilege to the appropriate role (Application role or invoker's rights role) ensures secure access. Embedding the key directly in code (B) or as an environment variable (D) is insecure. While a UDF retrieving the key (C) could work, it adds complexity and potential performance overhead compared to using Snowflake's native secret management.

NEW QUESTION: 109

You are developing a Snowflake Native Application that leverages user-defined functions (UDFs) and stored procedures. During the security scanning process before listing on the Snowflake Marketplace, you encounter a 'Failed to Detect Secrets' error. After investigation, you determine that the error stems from hardcoded API keys within a Python UDF that are being used to interact with a third-party service. Which of the following strategies would be MOST effective and secure to resolve this issue and pass the security scan?

- A. Encrypt the API keys within the Python UDF using a symmetric encryption algorithm and store the encryption key in a secure string within the application package.
- B. Replace the hardcoded API keys with references to Snowflake secrets and grant the necessary privileges to the application role. Access the secrets within the UDF using the 'SYSTEM\$GET SECRET' function.
- C. Obfuscate the API keys within the Python UDF using a string manipulation technique. This will make it harder for unauthorized users to discover the keys.
- D. Remove the Python UDF entirely and rewrite the logic in SQL stored procedures. Store the API keys as environment variables within the stored procedure's definition.
- E. Store the API keys in a separate, secured table within the application's data container and access them dynamically within the Python UDF using SQL queries.

Answer: (SHOW ANSWER)

Snowflake Secrets are the recommended and most secure way to handle sensitive information like API keys. Using 'SYSTEM\$GET_SECRET' allows you to retrieve the secrets at runtime without hardcoding them in the application code. Options A and C provide weak security and are easily compromised. Option D may not be feasible depending on the complexity of the UDF. Option E introduces complexities in data access and security management.

NEW QUESTION: 110

You are building a Streamlit application within a Snowflake Native App that allows users to download data as CSV. The data is prepared using a Snowflake stored procedure. The Streamlit app uses a Snowflake Connector for Python to call the stored procedure and retrieve results. Which of the following privileges are required for the 'app_public' role to successfully execute the stored procedure from within the Streamlit application? Select all that apply.

- A. USAGE on the database containing the stored procedure.
- B. EXECUTE on the stored procedure.
- C. OWNERSHIP on the stored procedure.

D. USAGE on the schema containing the stored procedure.

E. CREATE STORED PROCEDURE on the schema containing the stored procedure

Answer: ([SHOW ANSWER](#))

The correct answers are B and D. To execute a stored procedure, the 'app_public' role needs the 'EXECUTE privilege on the stored procedure itself (Option B), and 'USAGE on the schema (Option D) that contains the stored procedure. Option A is incorrect because USAGE is required on the schema, not database. Option C is incorrect because ownership isn't needed by the 'app_public' role, simply 'EXECUTE'. Option E is incorrect because 'CREATE STORED PROCEDURE is only required when creating stored procedures, not executing them.

NEW QUESTION: 111

You are developing a Snowflake Native Application that processes sensitive financial data. You need to implement robust logging and auditing to comply with regulatory requirements. Which of the following strategies would provide the MOST secure and comprehensive auditing and logging solution within the context of a Snowflake Native App?

A. Rely solely on Snowflake's ACCOUNT_USAGE views and information schema tables within the consumer account.

B. Implement custom logging using UDFs that write directly to a table accessible only within the provider account. Use masking policies to protect sensitive data before logging.

C. Utilize Snowflake's Event Tables to capture application events. Ensure proper role-based access control is in place to restrict access to the event data within the provider account. Masking and Row Access policies should be applied before writing to event tables.

D. Implement logging to an external service such as AWS CloudWatch or Azure Monitor, streaming data directly from UDFs using external functions. Implement data redaction within the UDFs before sending data externally.

E. Use Snowflake's stages to write the event logs and then create the external table on top of it to process the data.

Answer: ([SHOW ANSWER](#))

Snowflake Event Tables provide a native, secure, and auditable way to track application events within the provider account. They are designed for observability and telemetry within the secure context of Snowflake. While options B and D are possible, they introduce complexities and potential security vulnerabilities related to managing access to external systems and ensuring data protection during transit. Option A provides insufficient application-specific logging. Option E, uses stages but not the snowflake event tables, stages are not the most suitable for application- specific data.

NEW QUESTION: 112

You are tasked with configuring a Snowflake Native Application that leverages Snowpark Container Services for a data transformation pipeline. The pipeline requires access to a public API to enrich data. However, the consumer's environment has strict network egress policies. The API endpoint is 'https://api.example.com/v1/data'. Which combination of Snowflake objects and configurations would best enable the container service to access this external API while adhering to the security policies ?

A. Create a Snowflake external function that calls the API. Then, call the external function from within the Snowpark Container Service.

B. Configure a Snowflake network rule specifying 'https://api.example.com/v1/data' as the allowed egress endpoint. Associate this rule with a network policy and assign the network policy to the container service.

C. Create a Snowflake stage with external access integration pointing to 'https://api.example.com/v1/data'. Mount this stage in the container service.

D. Use the role to bypass network policies and directly access the API from the container service.

E. Configure an API integration in Snowflake and then use it within the Snowpark Container Service to authenticate and access the API.

Answer: ([SHOW ANSWER](#))

The correct approach is to use a Snowflake network rule and policy to explicitly allow egress traffic to the specified API endpoint. An external function is an option, but it adds unnecessary complexity for a simple API call. Snowflake stages are for storing data, not for making API calls. Bypassing security policies with 'SNOWFLAKE CONTAINER SERVICE is incorrect and insecure. API integrations in Snowflake are used for securing access to Snowflake data and resources from external services, not the other way around. Therefore, creating a network rule and associating it with the container service's network policy is the best option.

NEW QUESTION: 113

A Snowflake Native Application is being developed. The application requires access to a third-party API to enrich data. To securely store the API key, the developer plans to use a Secret object within Snowflake. Which of the following steps are necessary to grant the application, specifically an application role named 'data_processor' , the ability to read this Secret object? (Choose three)

- A. Grant the OWNERSHIP privilege on the Secret object to the 'data_processor' application role.
- B. Grant the USAGE privilege on the secret object to the 'APPLICATION PACKAGE'.
- C. Grant the READ privilege on the Secret object to the 'data_processor' application role.
- D. Grant the USAGE privilege on the Secret object to the 'data_processor' application role.
- E. The 'data_processor' application role automatically inherits privileges for accessing Secrets defined within the application package.

Answer: ([SHOW ANSWER](#))

To allow an application role to access a Secret object, you must explicitly grant the READ privilege on the secret to that application role. Additionally, you must also explicitly grant the USAGE privilege on the secret to that application role. OWNERSHIP grants all privileges and is not necessary in this scenario. Granting privileges to the APPLICATION PACKAGE only allows those privileges to be granted to application roles, not directly used by the application.

NEW QUESTION: 114

A consumer installs a Snowflake Native App that performs data transformations. The consumer wants to monitor the resource consumption (e.g., credits used) of the application. Which of the following SQL queries can the consumer use to retrieve the warehouse- level credit usage information associated with the installed application 'my_app'?

- A. SELECT start_time, end_time, warehouse_name, credits_used FROM snowflake.account_usage.warehouse_metering_history WHERE application_name = 'my_app';
- B. SELECT start_time, end_time, warehouse_name, credits_used FROM snowflake.account_usage.metering_history WHERE application_name = 'my_app';
- C. SELECT start_time, end_time, warehouse_name, credits_used FROM snowflake.account_usage.query_history WHERE application_name = 'my_app';
- D. SELECT start_time, end_time, warehouse_name, credits_used FROM snowflake.account_usage.resource_monitors WHERE application_name = 'my_app';
- E. SELECT start_time, end_time, warehouse_name, credits_used FROM snowflake.account_usage.database_storage_usage_history WHERE application_name = 'my_app';

Answer: ([SHOW ANSWER](#))

The 'snowflake.account_usage.warehouse_metering_history' view provides warehouse-level credit usage information. Filtering by application_name = 'my_app' allows the consumer to see the credit consumption specifically for the installed application. 'metering_history' contains overall account usage. 'query_history' contains query details, not credit usage. 'resource_monitors' are for setting limits, not usage. 'database_storage_usage_history' shows storage usage, not credit consumption for compute.

NEW QUESTION: 115

Consider the following setup script snippet for a Snowflake Native Application:

```
CREATE SCHEMA IF NOT EXISTS app_data;
CREATE TABLE IF NOT EXISTS app_data.configuration (
  param_name VARCHAR,
  param_value VARCHAR
);
INSERT INTO app_data.configuration (param_name, param_value)
SELECT 'api_endpoint', 'https://default.api.example.com'
WHERE NOT EXISTS (SELECT 1 FROM app_data.configuration WHERE param_name = 'api_endpoint');
```

What is the primary purpose of the 'WHERE NOT EXISTS' clause in the 'INSERT' statement within this setup script?

- A. To ensure that the 'INSERT' statement only executes if the table is empty.
- B. To prevent duplicate rows with the same 'param_name' from being inserted into the 'app_data.configuration' table during subsequent installations or upgrades of the application.
- C. To optimize the performance of the 'INSERT' statement by avoiding unnecessary insertions when the configuration parameter already exists.
- D. To handle potential errors that may occur if the 'api_endpoint' parameter is already defined in the consumer's account.
- E. To guarantee that the 'app_data.configuration' table always contains the latest version of the 'api_endpoint' parameter, overwriting any existing value.

Answer: ([SHOW ANSWER](#))

Option B is correct. The 'WHERE NOT EXISTS' clause prevents duplicate entries for the 'api_endpoint' parameter. The setup script might run multiple times during application installations or upgrades, and without this clause, it would insert duplicate rows. Option A is incorrect, as the clause doesn't check if the entire table is empty. Option C is true to some extent, performance is one of the reasons but not the primary reason. Option D, The consumer account has no role to play here. Option E is wrong because it will do opposite as intended.

NEW QUESTION: 116

You are developing a Snowflake Native App that processes streaming data from a Kafka topic. The app needs to continuously ingest data, perform real-time transformations, and store the results in a Snowflake table within the app's managed database. Which Snowflake feature(s) and approach(es) are BEST suited for implementing this data ingestion and processing pipeline within the Native App?

- A. Use Snowflake Pipes to continuously ingest data from the Kafka topic into a staging table and then use stored procedures to periodically transform and load the data into the target table.
- B. Leverage Snowflake Snowpipe Streaming to ingest data directly from the Kafka topic into the target table, combined with Snowflake Tasks to handle any required data transformations.
- C. Implement a custom Kafka consumer within a Java User-Defined Function (UDF) to read data from the Kafka topic and directly insert it into the target table.
- D. Use a Snowflake Stream to capture changes in the Kafka topic and then use a Snowflake Task to periodically merge these changes into the target table.
- E. Employ a combination of Snowflake Snowpipe Streaming for data ingestion and Dynamic Tables for continuous, incremental data transformations within the application's managed database.

Answer: ([SHOW ANSWER](#))

Snowpipe Streaming offers the lowest latency and near real-time data ingestion capabilities. Dynamic Tables provide a declarative and managed way to perform continuous data transformations, making them ideal for processing streaming data within a Snowflake Native App. Options A, B, and C are less suitable for high-velocity streaming data due to latency and complexity concerns. Streams are for tracking changes within Snowflake tables, not external Kafka topics.

NEW QUESTION: 117

You are packaging a Snowflake Native App that includes a Streamlit application for data visualization. The Streamlit application needs to access a table named 'SALES DATA' within the app's data container. Which of the following steps are REQUIRED to correctly grant the Streamlit application the necessary permissions to read the data? (Select all that apply)

- A. Create a service account within the Snowflake Native App and grant the 'SELECT' privilege on the 'SALES_DATA' table to the service account.
- B. Grant the 'APPLOG' privilege to the application role defined in the setup script on the internal stage.
- C. Create a secure view on the 'SALES_DATA' table, using the function in the view definition, and grant the 'SELECT' privilege on the secure view to the application role.
- D. Grant the 'SELECT' privilege on the 'SALES_DATA' table directly to the application role defined in the setup script.
- E. Grant the 'IMPORTED PRIVILEGES' on the database containing the 'SALES DATA' table to the application role.

Answer: ([SHOW ANSWER](#))

Snowflake Native Apps require service accounts for components like Streamlit to interact with data securely. Granting SELECT directly to the app role (D) is generally not how applications are structured with secure views. APPLOG (B) is related to logging, not data access. IMPORTED PRIVILEGES (E) is useful when referencing objects from external data sources which is not the case here, the table resides within the application's data container. Using a secure view (C) allows for controlled access and data masking if needed and service account will be used to access it.

NEW QUESTION: 118

A Snowflake Native App developer is packaging their application and needs to define the necessary privileges within the application package's manifest file. The application requires 'SELECT' access to a view named 'PROVIDER VIEW' and 'USAGE' on the database 'PROVIDER DB'. Which of the following YAML snippets correctly defines these privileges in the manifest file? (Choose two)

```
resources:
  - name: PROVIDER_VIEW
    type: view
    privileges: [SELECT]
  - name: PROVIDER_DB
    type: database
    privileges: [USAGE]
```

A.

- B.

```
privileges:
- resource: PROVIDER_VIEW
  actions: [SELECT]
- resource: PROVIDER_DB
  actions: [USAGE]
```
- C.

```
resources:
- name: PROVIDER_VIEW
  type: VIEW
  privileges: [SELECT]
- name: PROVIDER_DB
  type: DATABASE
  privileges: [USAGE]
```
- D.

```
permissions:
- object: PROVIDER_VIEW
  type: VIEW
  access: SELECT
- object: PROVIDER_DB
  type: DATABASE
  access: USAGE
```
- E.

```
grants:
- resource_name: PROVIDER_VIEW
  resource_type: view
  privilege: SELECT
- resource_name: PROVIDER_DB
  resource_type: database
  privilege: USAGE
```

Answer: (SHOW ANSWER)

Options A and C are functionally equivalent and the only two valid configurations. Snowflake manifest files use the 'resources' section to define the objects and their required privileges. Note that Snowflake resource types (like VIEW and DATABASE) are case-insensitive. Options B, D, and E do not conform to the expected manifest file structure for defining resources and their associated privileges in a Snowflake Native App. The actual syntax and keyword usages are crucial for correct interpretation by the Snowflake platform. The only difference between A and C is that it does not affect functionality but will use the best format.

NEW QUESTION: 119

You are preparing to list your Snowflake Native Application on the Snowflake Marketplace. You've developed the application and implemented security measures. Which of the following steps are essential best practices for ensuring a successful and secure listing on the Snowflake Marketplace?

- A. Thoroughly document the application's functionality, installation instructions, configuration options, and any known limitations. Provide clear and concise pricing information and support contact details.
- B. Implement robust error handling and logging mechanisms to provide consumers with informative error messages and aid in troubleshooting. Include mechanisms for securely collecting and storing application usage metrics (if required), ensuring compliance with privacy regulations.
- C. Clearly communicate the application's security model, including data encryption methods, access controls, and any reliance on consumer-side security configurations. Undergo and document regular security audits and penetration tests of the application.

- D.** Include a prominently displayed disclaimer stating that the application is provided "as is" and the provider assumes no liability for any damages arising from its use. Only provide basic installation instructions. It is the end user's responsibility to know how to use the application.
- E.** Avoid documenting any limitations or potential issues with the application to present a flawless image to potential consumers. Only provide limited pricing information to create a sense of urgency to potential consumer

Answer: ([SHOW ANSWER](#))

Options A, B, and C are essential best practices. Thorough documentation (A) is crucial for consumer adoption. Robust error handling and logging (B) improve the user experience and facilitate troubleshooting. Clear communication of the security model (C) builds trust. Option D is unacceptable as it shifts responsibility unfairly to the consumer. Option E is also unethical, as consumers should know what to expect.

NEW QUESTION: 120

You are publishing a Snowflake Native Application to the Marketplace. You've completed initial testing, and the application appears to function as expected. However, the security scan consistently fails with a 'Data Exfiltration' error. The application includes a stored procedure that aggregates data from several tables within the application container and writes the results to an external stage using the 'COPY INTO' command. Review the following code snippet:

```
CREATE OR REPLACE PROCEDURE export_data(stage_name VARCHAR) RETURNS VARCHAR LANGUAGE SQL AS $$
BEGIN
  COPY INTO @" + stage_name + "/output.csv
  FROM (SELECT FROM aggregated_data);
  RETURN 'Data exported successfully';
END;
$$;
GRANT EXECUTE ON PROCEDURE export_data(VARCHAR) TO APPLICATION ROLE app_public;
```

Which of the following changes is MOST likely to resolve the 'Data Exfiltration' error during the security scan?

- A.** Remove the 'GRANT EXECUTE privilege on the 'export_data' procedure from the 'app_public' role.
- B.** Implement row-level security policies on the 'aggregated_data' table to restrict access to sensitive rows.
- C.** Require the consumer to explicitly grant the application usage privilege on a secure external stage controlled by the consumer, and modify the procedure to validate the stage's existence and permissions before exporting data.
- D.** Encrypt the data written to the external stage using client-side encryption with a key managed by the application provider.
- E.** Modify the stored procedure to only export data to a table within the application container instead of an external stage.

Answer: ([SHOW ANSWER](#))

The 'Data Exfiltration' error indicates a potential risk of unauthorized data leaving the consumer's Snowflake environment. The root cause is often that the application provider has unrestricted control over where the data is exported. The solution is to require the consumer to control the external stage and explicitly grant the application usage privilege on it. This ensures that the consumer is aware of and approves the data export. Option A would break the functionality. Option B addresses data access within the container but doesn't prevent exfiltration if the application can still write to an uncontrolled stage. Option D adds complexity but doesn't fundamentally address the control issue. Option E restricts functionality unnecessarily.

NEW QUESTION: 121

You are building a Snowflake Native App that needs to store configuration settings specific to each consumer account. These settings will influence the behavior of the application. Which of the following methods is the MOST secure and recommended approach for managing these configuration settings within the Snowflake Native App Framework?

- A.** Store the configuration settings directly within the application's code. This allows for easy access and modification during development.
- B.** Utilize parameters defined in the 'setup.sql' script and accessible via SQL commands like 'SYSTEM\$GET APP PARAMETER'. This allows consumers to configure the application during installation.
- C.** Hardcode the configuration settings directly into the manifest file as environment variables.
- D.** Store the configuration settings in a public database table within the application's share. This makes the settings easily accessible to all consumers.
- E.** Use external configuration files stored in an Amazon S3 bucket and accessed via external functions. This provides flexibility and scalability.

Answer: ([SHOW ANSWER](#))

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 122

You are creating a Snowflake Native Application that will be distributed to consumer accounts. As part of the installation process, you need to ensure that a specific database role, 'DATA READER, is created in the consumer account and granted 'SELECT' privileges on a specific table in the consumer's shared dat a. Which of the following code snippets, when placed within the application's setup script (install.sql), will correctly achieve this, assuming the application is granted 'CREATE DATABASE ROLE?

- CREATE OR REPLACE DATABASE ROLE DATA_READER;
GRANT SELECT ON TABLE CONSUMER_DB.PUBLIC.MY_TABLE TO DATABASE ROLE DATA_READER;
GRANT DATABASE ROLE DATA_READER TO SHARE;
- A. CREATE OR REPLACE DATABASE ROLE DATA_READER;
GRANT SELECT ON TABLE CONSUMER_DB.PUBLIC.MY_TABLE TO DATABASE ROLE DATA_READER;
GRANT DATABASE ROLE DATA_READER TO APPLICATION ROLE app_public;
- B. CREATE OR REPLACE DATABASE ROLE DATA_READER;
GRANT SELECT ON TABLE CONSUMER_DB.PUBLIC.MY_TABLE TO DATABASE ROLE DATA_READER;
GRANT DATABASE ROLE DATA_READER TO APPLICATION ;
- C. CREATE OR REPLACE DATABASE ROLE DATA_READER;
GRANT SELECT ON TABLE CONSUMER_DB.PUBLIC.MY_TABLE TO DATABASE ROLE DATA_READER;
GRANT DATABASE ROLE DATA_READER TO APPLICATION ROLE app_public; GRANT APPLICATION ROLE app_public TO APPLICATION ;
- D. CREATE OR REPLACE DATABASE ROLE DATA_READER;
GRANT SELECT ON TABLE CONSUMER_DB.PUBLIC.MY_TABLE TO DATABASE ROLE DATA_READER;
GRANT DATABASE ROLE DATA_READER TO APPLICATION ROLE app_public;
GRANT APPLICATION ROLE app_public TO APPLICATION ;
GRANT APPLICATION ROLE app_public TO SHARE;
- E.

Answer: ([SHOW ANSWER](#))

Option D correctly creates the database role, grants the necessary privileges, and associates it with the application. Here's a breakdown: 'CREATE OR REPLACE DATABASE ROLE DATA_READER;': creates or replaces the database role 'DATA_READER in the consumer account. 'GRANT SELECT ON TABLE CONSUMER DB.PUBLIC.MY TABLE TO DATABASE ROLE DATA READER;': Grants the 'SELECT privilege on the specified table to the 'DATA_READER role. It assumes that already exists in the consumer account through data sharing or some other mechanism. 'GRANT DATABASE ROLE DATA READER TO APPLICATION ROLE app_public;': This grants the database role to the application role, making the database role available to the application via the 'app_public' application role. 'GRANT APPLICATION ROLE app_public TO APPLICATION This is essential. This makes sure the application has the privileges from the application role granted. Without this, the database role is effectively useless to the application. In the consumer account, this means it will be able to use application. It should have an application object representing installed app in Consumer Account. Option A is incorrect because granting to 'SHARE is not the appropriate way to grant the role to the application. This is meant for granting to other accounts. Option B is incorrect because application will not be able to use app_public if not granted. Option C is incorrect because you need to assign database role to application role so it can be properly managed at the application level. You also need to grant the application role to the application Option E is incorrect because you should not grant the application role to SHARE; it will provide access to any accounts using the share, and can lead to over-privileged access.

NEW QUESTION: 123

You are developing a Snowflake Native Application that interacts with sensitive customer data. You want to implement robust security scanning before publishing to the Snowflake Marketplace. Which of the following strategies are MOST effective to detect and prevent potential vulnerabilities before submitting your application, considering limitations on marketplace pre-publication scanning capabilities?

- A.** Rely solely on Snowflake's built-in scanning during the Marketplace submission process, as it covers all known vulnerabilities.
- B.** Implement a comprehensive CI/CD pipeline that includes static code analysis (SAST) using tools like SonarQube or similar, focusing on SQL injection, cross-site scripting (XSS), and insecure deserialization vulnerabilities. Include unit and integration tests with parameterized inputs. Regularly update dependency versions and scan for known vulnerabilities in third-party libraries included in your application's code (e.g., Python libraries).
- C.** Manually review all code for potential security flaws and ensure adherence to secure coding practices.
- D.** Employ a third-party penetration testing service to perform a black-box test of the application after deployment in a staging environment but prior to Marketplace submission.
- E.** Only use Snowflake provided templates and example codes as these are already pre-approved from security perspectives

Answer: ([SHOW ANSWER](#))

Option B and D provides a good answer as Snowflake's Marketplace scanning is limited, a multi-layered approach is crucial. SAST and dependency scanning within a CI/CD pipeline (B) allows for early detection of many common vulnerabilities. Penetration testing (D) provides an external, real-world assessment of the application's security posture. Relying solely on Snowflake's scanning (A) is insufficient. Manual code review (C), while helpful, is prone to human error. Only using Snowflake provided templates is very restrictive and not a solution.

NEW QUESTION: 124

A Snowflake Native Application is designed to process customer data using a series of stored procedures. One of the stored procedures, process data', requires access to a stage named 'customer stage' in the consumer's account to load data. The application role 'app_role' is intended to manage all access within the application. However, the procedure fails with a permission error during testing in the consumer environment. Which of the following combination of steps is required to fix the problem?

- A.** Grant 'USAGE privilege on the database and schema containing 'customer_stage' to the 'app_role', and grant 'READ privilege on 'customer_stage' to the app_role' .
- B.** Grant 'READ and ' WRITE privilege on 'customer_stage' directly to the provider account.
- C.** Grant 'USAGE privilege on the database and schema containing 'customer_stage' to the provider account, and grant privilege on 'customer_stage' to the 'app_role' .
- D.** Grant 'USAGE privilege on the database and schema containing 'customer_stage' to the 'app_role', and create a new role in the consumer account, grant 'READ on the 'customer_stage' to this new role, then grant this new role to 'app_role' .
- E.** The stored procedure needs to be defined with 'EXECUTE AS CALLER rights to assume the privileges of the role executing the procedure.

Answer: ([SHOW ANSWER](#))

Options A and E are correct. Option A: The 'app_role' needs 'USAGE privileges on the database and schema to access objects within them. It also needs 'READ' privilege on the stage to load data from it. Option E: If the stored procedure is accessing objects with the 'OWNER rights instead of 'CALLER rights, then it doesn't matter what the 'app_role' has privileges to. The procedure needs to be defined to execute as the caller to take advantage of the consumers permissions to external stages. Options B and C are incorrect because granting privileges directly to the provider account or the consumer account is the incorrect pattern for Snowflake Native Apps. The privileges should be managed within the consumer account, usually through an 'app_role' . Option D includes an unnecessary intermediary role that is not needed.

NEW QUESTION: 125

Your Snowflake Native App includes a JavaScript UDF that processes data from a secure view provided by the provider account. During consumer testing, the UDF fails with an 'Authorization error' even though the consumer has been granted the necessary USAGE privileges on the application package. What is the MOST likely cause of this error, and how can you resolve it within the context of a Snowflake Native App?

- A.** The consumer lacks the necessary permissions on the underlying tables referenced by the secure view in the provider account. Grant the consumer direct access to these tables.
- B.** The UDF is not declared as SECURE, preventing it from accessing objects in a different security context. Declare the UDF as SECURE.
- C.** The application package needs to be explicitly authorized to access secure views in the provider account. This requires a manifest update and resubmission of the package.
- D.** The secure view in the provider account has data masking policies applied that prevent the UDF from accessing the data. Remove the masking policies.
- E.** The consumer's role does not have the OWNERSHIP privilege on the application object. Assign the OWNERSHIP privilege to the consumer's role.

Answer: ([SHOW ANSWER](#))

The most likely cause is that the UDF needs to be declared as 'SECURE' (Option B). When a UDF accesses objects in a different security context (such as a secure view provided by the provider), it must be declared as SECURE to inherit the privileges of the caller (in this case, the application). This allows the UDF to access the data without requiring the consumer to have direct access to the underlying tables. A: Granting direct access to the underlying tables defeats the purpose of using a secure view and is generally not recommended. C: There is no explicit authorization needed in the manifest for a native app to access secure views if the secure UDF accesses the view D: While data masking policies could cause issues, the 'Authorization error' specifically points to a privilege problem, making the SECURE UDF the more likely culprit. E: The OWNERSHIP privilege on the application is not relevant to the UDF's ability to access data within the provider account.

NEW QUESTION: 126

A Snowflake Native App provider wants to ensure that even future roles created in their provider account will NOT automatically have access to the application database , which was created without the "MANAGED ACCESS" property initially. After creating a security audit suggested it to be with managed access. Which of the following steps BEST secures the app database?

Note multiple options may be needed

- A. DROP DATABASE CREATE DATABASE WITH MANAGED ACCESS;
- B. ALTER DATABASE SET MANAGED ACCESS = TRUE;
- C. REVOKE ALL PRIVILEGES ON DATABASE FROM PUBLIC;
- D. REVOKE USAGE ON DATABASE FROM ROLE PUBLIC;
- E. GRANT OWNERSHIP ON DATABASE TO ROLE application_role;

Answer: B,C,E (LEAVE A REPLY)

The most effective method is to use 'ALTER DATABASE SET MANAGED_ACCESS = TRUE;' to enable managed access. This ensures that privileges are managed at the database level. Revoking all privileges from PUBLIC removes default access granted to all roles (including future roles). By revoking default public access and using managed access together, one ensures greater security. Option A works, but dropping and recreating is impactful. Option D, is similar to C however doesn't handle future role. Granting ownership to a specific role ensures better control.

NEW QUESTION: 127

You are preparing to list your Snowflake Native Application on the Snowflake Marketplace. You want to restrict its availability to specific Snowflake regions and also require consumers to agree to a custom End User License Agreement (EULA) before installation. Which steps are essential during the listing creation process to achieve this?

- A. Configure the 'Availability' section of the listing to specify allowed regions. Upload the EULA as a PDF document within the 'Documentation' section.
- B. Use the Snowflake CLI to programmatically set the allowed regions using the 'snowflake marketplace application update' command. Include a link to an external website hosting the EULA in the listing description.
- C. Specify the allowed regions in the 'Availability' section of the listing. Upload a text file containing the EULA and mark it as 'Required Agreement' in the listing settings.
- D. During listing creation, select the 'Custom EULA' option and provide the EULA text directly within the designated field. Configure the 'Availability' section to specify the allowed regions.
- E. Regions are inherently globally available so it cannot be restricted. Include the EULA within the code of the application, forcing the user to acknowledge it through the code.

Answer: (SHOW ANSWER)

Option D is the correct answer. The Snowflake Marketplace provides specific fields to enter a custom EULA directly. Also the availability of the application can be restricted based on Snowflake regions. Options A, B, and C provide partially correct steps, but do not cover the complete solution. A PDF or link doesn't guarantee agreement. Option E has a false statment that regions cannot be restricted.

NEW QUESTION: 128

You are developing a Snowflake Native Application that requires fine-grained control over data access for its users. You plan to use Streamlit within the application and leverage the Permissions API to manage these privileges. Assume you have successfully established a connection to Snowflake within your Streamlit app. Which of the following code snippets accurately demonstrate how to grant the 'SELECT privilege on a table named 'APP DATA in the application's schema to a specific role, 'APP USER, using the Permissions API, and then verify if that privilege has been granted?

- import snowflake.connector import streamlit as st ctx = snowflake.connector.connect(st.secrets["snowflake"]) cur = ctx.cursor() role_name = 'APP_USER' schema_name = 'APP_SCHEMA' table_name = 'APP_DATA' # Grant SELECT privilege cur.execute(f"GRANT SELECT ON TABLE {schema_name}.{table_name} TO ROLE {role_name}") # Verify if privilege is granted (Incorrect Method) cur.execute(f"SHOW GRANTS TO ROLE {role_name}") results = cur.fetchall() for row in results: if row[5] == 'SELECT' and row[3] == table_name and row[2] == schema_name: st.write(f"SELECT privilege granted on {schema_name}.{table_name} to role {role_name}") break
- A. role {role_name}') break
- B. ○ import snowflake.connector import streamlit as st from snowflake.permissions import PermissionsClient ctx = snowflake.connector.connect(st.secrets["snowflake"]) cur = ctx.cursor() role_name = 'APP_USER' schema_name = 'APP_SCHEMA' table_name = 'APP_DATA' # Grant SELECT privilege (Incorrect Method) cur.execute(f"GRANT SELECT ON TABLE {schema_name}.{table_name} TO ROLE {role_name}") # Verify if privilege is granted cur.execute(f"SHOW GRANTS TO ROLE {role_name}") results = cur.fetchall() for row in results: if row[3] == 'SELECT' and row[5] == table_name: st.write(f"SELECT privilege granted on {schema_name}.{table_name} to role {role_name}") break
- import snowflake.connector import streamlit as st from snowflake.permissions import PermissionsClient ctx = snowflake.connector.connect(st.secrets["snowflake"]) cur = ctx.cursor() role_name = 'APP_USER' schema_name = 'APP_SCHEMA' table_name = 'APP_DATA' # Grant SELECT privilege using Permissions API (Hypothetical) permissions_client = PermissionsClient(ctx) permissions_client.grant_select(table_name, schema_name, role_name) # Verify if privilege is granted (Hypothetical) is_granted = permissions_client.check_privilege('SELECT', table_name, schema_name, role_name) if is_granted: st.write(f"SELECT privilege granted on {schema_name}.{table_name} to role {role_name}")
- C. ○ import snowflake.connector import streamlit as st ctx = snowflake.connector.connect(st.secrets["snowflake"]) cur = ctx.cursor() role_name = 'APP_USER' schema_name = 'APP_SCHEMA' table_name = 'APP_DATA' # Grant SELECT privilege cur.execute(f"GRANT SELECT ON {schema_name}.{table_name} TO ROLE {role_name}") # Verify if privilege is granted (using a highly specific query) cur.execute(f"SELECT COUNT() FROM INFORMATION_SCHEMA.TABLE_PRIVILEGES WHERE GRANTEE = '{role_name}' AND TABLE_NAME = '{table_name}' AND TABLE_SCHEMA = '{schema_name}' AND PRIVILEGE = 'SELECT'") result = cur.fetchone()[0] if result > 0: st.write(f"SELECT privilege granted on {schema_name}.{table_name} to role {role_name}")
- D. ○ import snowflake.connector import streamlit as st ctx = snowflake.connector.connect(st.secrets["snowflake"]) cur = ctx.cursor() role_name = 'APP_USER' schema_name = 'APP_SCHEMA' table_name = 'APP_DATA' # Grant SELECT privilege cur.execute(f"GRANT SELECT ON TABLE {schema_name}.{table_name} TO ROLE {role_name}") # Verify if privilege is granted (using a more general query) cur.execute(f"SHOW GRANTS TO ROLE {role_name}") results = cur.fetchall() found = False for grant in results: if grant[2].upper() == schema_name.upper() and grant[3].upper() == table_name.upper() and grant[5].upper() == 'SELECT': found = True break if found: st.write(f"SELECT privilege granted on {schema_name}.{table_name} to role {role_name}") else: st.write(f"SELECT privilege NOT granted on {schema_name}.{table_name} to role {role_name}")
- E. granted on {schema_name}.{table_name} to role {role_name}')

Answer: (SHOW ANSWER)

Option E is correct because it accurately grants the SELECT privilege and then uses the 'SHOW GRANTS TO ROLE' command to verify the privilege. It also includes a check to ensure the schema, table, and privilege all match the expected values, handling upper/lower case variations. Option A is similar, but has inconsistent index access in the SHOW GRANT results, and its check might be too strict by requiring the role to match. Option B incorrectly accesses the results of SHOW GRANTS, checking the wrong indices. Option C mentions a hypothetical PermissionsClient which is NOT available in Snowflake. The Permissions API does not have 'grant_select' or 'check_privilege' methods like that. Option D uses INFORMATION_SCHEMA which is less reliable in the context of Snowflake Native Applications. The table name case-sensitivity matters with INFORMATION_SCHEMA. The Information Schema is dependent on the privileges of the session role.

NEW QUESTION: 129

You're packaging a Snowflake Native Application that performs complex analytical computations on data within the consumer's account. This computation involves several intermediate tables and functions, all created within the application package. You want to optimize the application's performance and reduce storage costs in the consumer's account. Which of the following strategies can be employed to achieve these goals, especially considering that intermediate data is not intended to be directly accessible to the consumer?

- A. Create all intermediate tables as temporary tables. Temporary tables are automatically dropped at the end of the session, minimizing storage costs and ensuring data privacy.
- B. Create all intermediate tables as transient tables. Transient tables do not support Time Travel or Fail-safe, which reduces storage costs, and grant appropriate privileges to application roles.
- C. Implement data pruning techniques within the application's logic to delete intermediate data as soon as it is no longer needed, even if permanent tables are used for intermediate steps.
- D. Use external tables to store the intermediate results in cloud storage (e.g., AWS S3), accessed through a secure integration. This minimizes the storage footprint within the consumer's Snowflake account.
- E. Create all intermediate tables as standard permanent tables, relying on Snowflake's automatic data compression and storage optimization features to minimize costs. Grant limited privileges to the consumer to control data access.

Answer: (SHOW ANSWER)

Options B and C are correct because they both contribute to optimizing performance and reducing storage costs without compromising data privacy. Transient tables (Option B) reduce storage costs by eliminating Time Travel and Fail-safe, while still allowing for persistence within the consumer's account. Data pruning (Option C) directly reduces the amount of data stored by deleting intermediate results as soon as they are no longer needed. Using temporary tables (Option A) might not be suitable if the application's computation spans multiple sessions or contexts. Using external tables (Option D) would introduce additional complexity and overhead related to data transfer and external storage management. Relying solely on Snowflake's automatic optimization features (Option E) might not be sufficient to minimize storage costs, especially for large intermediate datasets.

NEW QUESTION: 130

A Snowflake Native Application developer, Bob, is using Snowsight to manage access to the application. He has created a custom role, , and wants to grant this role the ability to grant and revoke privileges on all objects within the application's schema, app_data. Which of the following Snowsight actions (or sequence of actions) is the MOST effective and secure way to achieve this? (Assume Bob is logged in as a user with 'ACCOUNTADMIN' role).

- A. Within Snowsight, navigate to the 'app_data' schema, select 'Grant Privileges', and grant the 'OWNERSHIP' privilege to the .
- B. Within Snowsight, navigate to the 'app_data' schema, select 'Grant Privileges', grant the 'USAGE' privilege to the and then individually grant 'SELECT', 'INSERT', 'UPDATE' , and "DELETE privileges on each table and view within the schema.
- C. Grant the the 'APPLY MASKING POLICY' privilege globally at the account level using Snowsight.
- D. In Snowsight, write and execute the following SQL command:
- E. Within Snowsight, navigate to the application database, select 'Grant Privileges', and grant the 'USAGE' privilege to the .

Answer: (SHOW ANSWER)

Granting 'OWNERSHIP' on the schema to (Option A) is the most effective and secure way to give the role full control over the schema and its objects. 'OWNERSHIP' implicitly grants all other privileges and the ability to grant/revoke them to other roles. While Option D using 'GRANT ALL PRIVILEGES' seems similar, it doesn't automatically cascade to new objects created within the schema after the grant. Ownership does. Option B is tedious and not scalable. Option C is irrelevant to the task. Option E only grants access to the database, not control over the objects within.

NEW QUESTION: 131

You are preparing to publish a Snowflake Native Application on the Snowflake Marketplace. Your application requires several external functions (UDFs) that need to be securely called. Which of the following steps are crucial to correctly configure these external functions within your application package to ensure they function as expected after installation by consumers?

- A. Ensure that the external functions are defined using the 'SECURE' keyword and that the API integration object associated with the functions is included in the application package setup script.
- B. Grant the ' USAGE' privilege on the API integration object to the application role used by the consumer. This will be automatically inherited when the consumer installs the application.
- C. Provide clear documentation for the consumer to manually configure the API integration object in their Snowflake account after installing the application.
- D. Define external functions with 'VOLATILE' keyword to bypass security restrictions. As it needs to be executed during installation, define it in application setup scripts and the API integration object's details will be automatically configured at the consumer end.
- E. Set the application version as 'TRUSTED' using the 'ALTER APPLICATION VERSION' command after installation.

Answer: ([SHOW ANSWER](#))

External functions in Snowflake Native Applications require careful configuration to ensure secure operation. Using the 'SECURE keyword when defining the functions ensures that Snowflake enforces access controls. The API integration object needs to be included in the setup script so that the application environment can be automatically configured during installation on the consumer's account. Option B is incorrect because privileges aren't automatically inherited from the provider. Option C is undesirable as it puts burden on the consumer. Options D & E are incorrect and not valid approaches to solving the problem.

NEW QUESTION: 132

You are developing a Snowflake Native Application that provides data transformation services. You want to provide example SQL scripts to consumers within the application package to showcase how to use the application's functions. Where should these example scripts be stored to ensure they are accessible to consumers upon installation?

- A.** Store the example SQL scripts in a separate stage within the provider account and grant USAGE on the stage to the application role.
- B.** Store the example SQL scripts as files within the application package itself (e.g., in a 'scripts' directory) and use relative paths to access them from within the application code.
- C.** Store the example SQL scripts in a table within the application database and provide a view to access them.
- D.** Embed the example SQL scripts directly as string literals within the setup script.
- E.** Store the example SQL scripts in an external stage (e.g., AWS S3) and provide instructions to the consumer to load them.

Answer: ([SHOW ANSWER](#))

Storing the scripts within the application package (Option B) ensures that they are always available to the consumer as part of the application installation. Relative paths allow the application code to access them directly. Option A requires additional permissions setup. Option C is not suitable, it is not meant for storage of scripts like that. Option D makes the setup script excessively long and hard to maintain. Option E requires additional setup and is not self-contained.

NEW QUESTION: 133

You are developing a Snowflake Native Application for a data enrichment service and plan to list it on the Snowflake Marketplace. You want to control the costs incurred by consumers of your application while maximizing its accessibility. Which of the following strategies represent Snowflake Marketplace best practices for cost management and pricing?

- A.** Offer a free trial period with limited functionality to allow potential consumers to evaluate the application before committing to a paid subscription.
- B.** Implement a tiered pricing model based on data volume processed, compute resources consumed, or number of users accessing the application. Offer a free basic tier with limited functionality.
- C.** Provide detailed cost estimation documentation outlining the expected costs for different usage scenarios and data volumes.
- D.** Refrain from providing any cost-related information upfront to encourage broader adoption, deferring cost discussions until after installation and usage.
- E.** Use Snowflake's built-in resource monitors to proactively track application consumption and alert consumers when they approach predefined spending limits.

Answer: ([SHOW ANSWER](#))

Offering a free trial (A) allows users to evaluate. Tiered pricing (B) offers flexibility and accommodates different budgets. Providing cost estimation documentation (C) is transparent and helps users plan. Using resource monitors for alerts (E) enables proactive cost control. Hiding cost information (D) is not a best practice and can lead to negative user experiences and discourage adoption.

NEW QUESTION: 134

Your team is developing a Snowflake Native App that requires access to network resources outside of Snowflake. After configuring an external access integration, you encounter an error indicating that the application cannot access the specified endpoint, even though the integration appears to be correctly configured. Assuming all network policies are in place and the external access integration is correctly defined, what is the MOST likely cause of this issue and how can it be resolved?

- A.** The application role used by the application lacks the 'IMPORTED PRIVILEGES privilege on the external access integration. Grant this privilege using 'GRANT IMPORTED PRIVILEGES ON INTEGRATION TO APPLICATION ROLE
- B.** The application needs to be explicitly granted the USAGE privilege on the external access integration. Use 'GRANT USAGE ON INTEGRATION TO APPLICATION ROLE
- C.** The URL specified in the external access integration is incorrect or contains a typo. Verify the URL and ensure it matches the expected endpoint.
- D.** The network policy attached to the external access integration is too restrictive and is blocking access to the required endpoint. Review and update the network policy to allow access to the endpoint.

E. The external function or procedure attempting to use the integration has not been granted the REFERENCE USAGE privilege on the integration. Use 'GRANT REFERENCE USAGE ON INTEGRATION TO PROCEDURE ;

Answer: ([SHOW ANSWER](#))

The application role needs 'IMPORTED PRIVILEGES on the external access integration to utilize its settings (A). Also, the specific external functions or procedures within the application that are leveraging the external access integration require the 'REFERENCE USAGE privilege on the integration (E). Without both privileges granted, the application won't be able to access the external resources defined in the integration. Option B 'USAGE' is not the correct privilege for external access integrations. While options C and D could be potential issues, they are less likely if the integration appears to be correctly configured.

NEW QUESTION: 135

A Snowflake Native App developer wants to utilize a README.md file within their application package. Which of the following statements accurately describe the supported usage and purpose of the README.md file in this context?

- A. The README.md file is automatically displayed to consumers within the Snowflake Marketplace listing, providing a detailed overview of the app's functionality and installation instructions.
- B. The README.md file must adhere to a specific Snowflake schema for metadata to be automatically extracted and displayed within the consumer's Snowflake environment post-installation.
- C. The README.md file is a Markdown formatted file and used only for internal documentation by the app developer and is not exposed to consumers through any Snowflake interface.
- D. The README.md file can be programmatically accessed within the application logic using a Snowflake-provided API to dynamically generate documentation for the consumer.
- E. Snowflake automatically renders the README.md file as part of the application's user interface within the consumer's account, allowing them to view documentation directly from Snowflake.

Answer: ([SHOW ANSWER](#))

The README.md file in a Snowflake Native App package is primarily used to populate the app's listing in the Snowflake Marketplace. It provides information about the app to potential consumers. It's not used for internal documentation only, nor is it rendered within the consumer's account post-installation. It doesn't require a specific Snowflake schema, nor is it programmatically accessible from the app.

NEW QUESTION: 136

You are using the Snowflake Visual Studio Code (VS Code) extension to develop a Snowflake Native App. After making several changes, you encounter an error that seems to stem from a syntax issue. You want to quickly pinpoint the exact location of the error in your SQL code. What functionality of the VS Code extension can best assist you in this scenario?

- A. The extension's integrated debugger, which allows you to step through your SQL code line by line.
- B. The syntax highlighting and code completion features, which help identify syntax errors and suggest valid code snippets.
- C. The extension's integration with Snowflake's query profile tool, enabling you to analyze the query execution plan for errors.
- D. The built-in linter that automatically identifies syntax errors, style issues, and potential security vulnerabilities in your SQL code.
- E. Using Version Control tool integration

Answer: ([SHOW ANSWER](#))

The correct answers are B and D. Syntax highlighting (B) visually cues potential errors while typing. The built-in linter (D) provides real-time automated error detection. Options A and C are not directly designed for quickly finding syntax issues. Version control tool integration is used for code management but can't help in identifying syntax error.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 137

You are tasked with creating a new version (2.0) of a Snowflake Native Application. This new version introduces significant schema changes and requires data migration from the old schema in version 1.0 to the new schema. You need to write the 'setup.sql' script for version 2.0. Which of the following steps or considerations are MOST critical to ensure a smooth and successful upgrade for existing customers? (Select TWO)

- A. Include only the schema changes in 'setup.sql' without any data migration logic. Assume users will manually migrate the data.
- B. Use 'setup.sql' to perform a direct 'INSERT INTO ... SELECT FROM' operation to move data from the old tables to the new tables after the new tables are created.
- C. Before applying schema changes, backup the existing data using 'CREATE TABLE AS SELECT' to safeguard against data loss during the upgrade process.
- D. Implement robust error handling and logging within the 'setup.sql' script to capture any issues during schema migration or data transfer.
- E. Use external stages for data backup, and leverage Snowflake Data Sharing to provide access to the older version data for consumers after upgrade.

Answer: ([SHOW ANSWER](#))

Options C and D are critical for a smooth and successful upgrade. Option C: Backing up the data before making any changes is a crucial safeguard against data loss during the upgrade process. The 'CREATE TABLE AS SELECT' command provides an easy way to create a backup of the data before schema modifications. Option D: Implementing robust error handling and logging is essential to monitor the upgrade process and identify any potential issues during schema migration or data transfer. This allows for quick intervention and troubleshooting, minimizing downtime and ensuring data integrity. Option A is incorrect because relying on manual data migration is impractical and error-prone for users. Option B might work for simple data migrations, but it lacks error handling and rollback capabilities. Option E is good practice from a security perspective.

NEW QUESTION: 138

You are developing a Snowflake Native App that requires specific privileges to access data in the consumer's account. In your manifest file, you want to request these privileges. Which of the following statements best describes the correct way to request these privileges within the 'privileges' section of the manifest file?

- A. Privileges are automatically granted to the application based on the roles specified in the application code. No explicit declaration is needed in the manifest file.
- B. Use the 'allowed_roles' section in the manifest to list the roles that are allowed to access data in the consumer's account, Snowflake automatically grants necessary privileges to these roles.
- C. Specify a list of required privileges under the 'privileges' section, detailing the specific objects (databases, schemas, tables) and actions (SELECT, INSERT, UPDATE) the application needs to perform.
- D. Privileges are managed entirely through the application's setup scripts. The manifest file does not play a role in privilege management.
- E. You can only request privileges for future grants. Existing object grants are not supported in manifest file.

Answer: ([SHOW ANSWER](#))

The 'privileges' section of the manifest file is used to declare the specific privileges required by the application. This includes specifying the objects (databases, schemas, tables) and actions (SELECT, INSERT, UPDATE) that the application needs to perform in the consumer's account. Snowflake uses this information to manage access control and ensure that the application has the necessary permissions to function correctly.

NEW QUESTION: 139

A consumer installs a Snowflake Native App that collects usage statistics. The app uses an internal stage to store temporary data before aggregating it and sending it to the provider. The consumer notices that the internal stage is consuming a significant amount of storage and wants to understand if they are being charged for this storage. As a consumer, what level of visibility and control do you have over the storage costs associated with the internal stage created and managed by the installed Snowflake Native App?

- A. The consumer has full visibility into the stage's storage consumption through the Snowflake web interface and can directly manage the stage's lifecycle to control costs.
- B. The consumer has no visibility into the stage's storage consumption or control over its lifecycle. All costs are borne by the application provider.
- C. The consumer is billed for the storage used by the internal stage, but they have no direct visibility into its usage or control over its lifecycle. The provider manages the stage's lifecycle.
- D. The consumer can view the overall storage consumption of the application but cannot differentiate between the internal stage and other data stored by the application.
- E. The consumer can only see the stage's metadata but cannot see the data residing in it.

Answer: ([SHOW ANSWER](#))

As a consumer, you are billed for the storage used by the application's internal stage. However, you have no direct visibility into the stage's usage or control over its lifecycle. The application provider manages the stage and data within. The consumer is billed for the storage within their account that the application uses. They are unaware of the data details within the application but still responsible for charges.

NEW QUESTION: 140

A Snowflake Native App provider is designing an application. Select ALL of the necessary privileges a provider needs on an Application Package to successfully list, version and distribute the native application to consumer accounts?

- A. OWNERSHIP
- B. USAGE
- C. APPLY
- D. MONITOR
- E. IMPORTED PRIVILEGES

Answer: (SHOW ANSWER)

The provider requires OWNERSHIP to manage the application package and APPLY to create versions and authorization grants, and IMPORTED PRIVILEGES is a privilege granted TO the consumer from the provider, not required to be owned by the provider. MONITOR is more applicable to usage monitoring and debugging, not core distribution privileges.

NEW QUESTION: 141

You have listed a Snowflake Native Application on the Marketplace. Several customers have installed your application and are actively using it. You discover a critical bug that requires an immediate fix. What is the MOST effective approach to deploy this fix to all existing installations of your application with minimal disruption to the consumers?

- A. Update the listing on the Marketplace with the corrected version of the application. Snowflake will automatically push the update to all consumers during their next maintenance window.
- B. Create a new version of the application package with the fix. Consumers will be notified of the available update and can choose when to upgrade.
- C. Use the 'ALTER APPLICATION ... UPGRADE' command in the provider account to forcibly upgrade all consumer installations to the latest version.
- D. Create a new version of the application package with the fix. Then send a notification to all consumers with instructions on how to manually upgrade their applications to the new version.
- E. Unlist the current application version from the Marketplace and relist the new, corrected version. This will force consumers to reinstall the application.

Answer: (SHOW ANSWER)

The best approach is to create a new version of the application package (B). Snowflake provides a mechanism for consumers to be notified of available updates and allows them to choose when to upgrade, providing control and minimizing disruption. Option A is incorrect because Snowflake does not automatically push updates. Option C is not a valid command and forcefully upgrading is generally discouraged. Option D is less efficient as it relies on manual consumer action. Option E is the worst option, as it would cause significant disruption and data loss for consumers. The application provider creates a new version of the application and publishes that to the marketplace, making it available for consumers to upgrade their installations.

NEW QUESTION: 142

You are setting up an external stage to host your Snowflake Native Application Package on AWS S3. You've created an IAM role and Snowflake integration. However, when you try to create the application package with 'EXTERNAL STAGE', you receive an error indicating that Snowflake cannot access the S3 bucket. Which of the following could be the potential causes and solutions?

- A. The IAM role does not have sufficient permissions to access the S3 bucket. Solution: Update the IAM role's policy to grant 's3:GetObject' and 's3:ListBucket' permissions on the bucket.
- B. The Snowflake integration object is not correctly configured with the IAM role ARN. Solution: Verify the parameter in the integration object matches the IAM role ARN.
- C. The S3 bucket policy does not allow access from the Snowflake AWS account. Solution: Update the bucket policy to explicitly allow access from the Snowflake AWS account, using the IAM role's principal.
- D. The external stage URL is incorrect. Solution: Ensure the URL provided in the 'CREATE APPLICATION PACKAGE EXTERNAL STAGE' command is the correct S3 bucket URL.
- E. The SNOWFLAKE DEPLOYMENT role is not granted access to the Integration object. Solution: Grant USAGE privilege on the Integration to the SNOWFLAKE DEPLOYMENT role.

Answer: (SHOW ANSWER)

Several issues can prevent Snowflake from accessing the S3 bucket. A,B,C, and D all represent potential problems. If the IAM role lacks the necessary permissions (A), Snowflake won't be able to retrieve the application package. If the integration object has an incorrect IAM role ARN (B), Snowflake won't be able to assume the correct role. If the S3 bucket policy doesn't allow access from Snowflake's AWS account (C), access will be denied. An incorrect stage URL (D) will also prevent access. SNOWFLAKE_DEPLOYMENT role is not related to accessing Integrations, so option E is irrelevant here.

NEW QUESTION: 143

A provider is developing a Snowflake Native Application that shares event data with consumers. The provider wants to ensure that specific columns in the event data are obfuscated for certain consumer accounts, while remaining visible to others. Which of the following approaches is the MOST secure and maintainable way to achieve this?

- A.** Create a separate view for each consumer account with the sensitive columns masked using a CASE statement based on the CURRENT_ACCOUNT() function.
- B.** Implement dynamic data masking policies on the sensitive columns using the CURRENT_ACCOUNT() function to determine which accounts receive obfuscated data.
- C.** Use a stored procedure to query the event data, applying obfuscation logic based on the consumer's APP_INSTANCE_ID() before returning the results.
- D.** Implement conditional logic within the event table function to obfuscate data based on the consumer's APP_INSTANCE_ID().
- E.** Data masking is not supported in Native Apps. The provider must rely on consumer-side masking implementations.

Answer: ([SHOW ANSWER](#))

B is the correct answer. Dynamic data masking policies offer a scalable and maintainable solution. By using the or function within the masking policy's expression, you can selectively mask columns based on the consumer account. This eliminates the need for creating separate views or complex stored procedures. A is not scalable and difficult to maintain. C & D are more complex than B and potentially less secure. E is wrong. Data masking is supported.

NEW QUESTION: 144

You are developing a Snowflake Native Application using Scala for a financial institution. The application processes sensitive transaction data. During consumer-side testing, the consumer reports intermittent 'java.lang.NullPointerException' errors. You suspect the issue lies within your UDF. How can you effectively trace and debug this issue, considering data privacy requirements and without directly exposing sensitive consumer data in your logs?

- A.** Use Snowflake's event table to capture all exceptions, filtering for the UDF name. Redact potentially sensitive fields in the logs after extraction using a Python script executed on the provider side.
- B.** Implement structured logging within the Scala UDF using SLF4J and Logback. Include contextual information (e.g., transaction IDs, timestamps) at various stages of the UDF execution. Use Snowflake's masking policies on the event table to protect sensitive data during logging. This is the best method as it's the more granular and effective.
- C.** Rely solely on Snowflake's query history to identify failing queries. Analyze the input data and execution plan to infer the cause of the NullPointerException.
- D.** Utilize 'System.out.println' statements within the Scala UDF to print debug messages directly to the query result. The consumer can view these messages to identify the source of the error, with the provider having no visibility.
- E.** Implement a custom error handling mechanism in Scala that catches 'NullPointerException', logs a generic error message with a unique error code to the event table, and then re-throws a more descriptive exception to the consumer. The consumer can then report the error code to the provider for further investigation. The provider can analyze the code, without actual consumer sensitive data.

Answer: ([SHOW ANSWER](#))

Options B and E provide effective and secure debugging strategies. B involves structured logging with masking policies to protect sensitive data, while E uses error codes for communication between consumer and provider without exposing raw data. Option A suggests post-extraction redaction which could be inefficient and prone to errors. Option C provides an insufficient approach. Option D is incorrect, and should never be used.

NEW QUESTION: 145

You are developing a Snowflake Native Application that uses Streamlit for its user interface. The application needs to access a custom Python module named 'my_module' which resides within your application package. You have correctly packaged the module. However, when running the Streamlit application, you encounter an 'ImportError: No module named' What is the most likely reason for this error and how can you resolve it?

- A.** The 'my_module' is not included in the file for your application package. Add it to the 'packages' list in 'setup.py'.
- B.** The Streamlit application is not aware of the application package's Python environment. You need to set the environment variable within the Streamlit application to point to the directory containing 'my_module'.
- C.** The 'my_module' is not located in the correct directory structure within the application package. Ensure it's placed in the 'src' directory (or the directory specified in 'package-name' in 'setup.py') and the directory structure mirrors the package structure required for import.
- D.** Streamlit does not support custom Python modules within Snowflake Native Applications. You must use only built-in Python modules or those available through Anaconda.
- E.** The correct handler is not specified in the 'manifest.yml' file.

Answer: ([SHOW ANSWER](#))

The most likely reason for the 'ImportError' is that the Python module is not located in the correct directory structure within the application package. When you build your application package, Snowflake expects your Python code to reside in a specific directory structure, typically the 'src' directory (or as configured in 'setup.py'). If the module is not in this structure or the correct package structure is not present, the Streamlit application will not be able to find it during import. Option A is incorrect, setup.py is for installation not direct packaging. Option B setting PYTHONPATH is not supported. Option D is incorrect, Custom Modules are supported by Streamlit. Option E is incorrect, as it has nothing to do with the module location and packaging.

NEW QUESTION: 146

A data science company is developing a Snowflake Native App that provides advanced fraud detection for financial institutions. The app requires secure access to sensitive transaction data within the consumer's Snowflake account. Which of the following approaches BEST aligns with Snowflake's recommended practices for managing data access and securing data within a Native App?

- A. The app should request full ACCOUNTADMIN privileges in the consumer account to access all necessary data for fraud detection.
- B. The app should use secure views created by the consumer that grant the app read-only access to only the specific transaction data needed for fraud detection.
- C. The app should directly access tables containing transaction data using the consumer's own service account credentials, which are securely stored within the app's code.
- D. The app should create and manage its own temporary tables in the consumer account and load transaction data into them for processing.
- E. The app should directly access transaction data through data sharing from provider account.

Answer: ([SHOW ANSWER](#))

Using secure views is the best approach. Secure views allow the consumer to control precisely what data the application can access without granting overly broad permissions. This minimizes the risk of unintended data exposure and adheres to the principle of least privilege. ACCOUNTADMIN is excessive and insecure. Storing credentials is also insecure and against best practices. Creating temporary tables and loading data duplicates data and can be inefficient.

NEW QUESTION: 147

A Native Application provider is developing an application that relies on data residing in a shared database, 'EXTERNAL DATA'. They want to ensure that their application role 'app_role' only has access to specific tables and views within 'EXTERNAL DATA', and they also want to minimize the need for future privilege grants as new tables are added to 'EXTERNAL DATA'. Which combination of SQL statements below achieves these goals with the principle of least privilege in mind? Assume all tables are in the PUBLIC schema.

- A.

```
GRANT SELECT ON FUTURE TABLES IN DATABASE EXTERNAL_DATA TO APPLICATION ROLE app_role; GRANT IMPORTED PRIVILEGES ON DATABASE EXTERNAL_DATA TO APPLICATION ROLE app_role;
```
- B.

```
GRANT USAGE ON DATABASE EXTERNAL_DATA TO APPLICATION ROLE app_role; GRANT SELECT ON ALL TABLES IN DATABASE EXTERNAL_DATA TO APPLICATION ROLE app_role;
```
- C.

```
GRANT IMPORTED PRIVILEGES ON DATABASE EXTERNAL_DATA TO APPLICATION ROLE app_role; GRANT SELECT ON FUTURE TABLES IN SCHEMA EXTERNAL_DATA.PUBLIC TO APPLICATION ROLE app_role;
```
- D.

```
GRANT USAGE ON DATABASE EXTERNAL_DATA TO APPLICATION ROLE app_role; GRANT SELECT ON FUTURE TABLES IN SCHEMA EXTERNAL_DATA.PUBLIC TO APPLICATION ROLE app_role;
```
- E.

```
GRANT IMPORTED PRIVILEGES ON DATABASE EXTERNAL_DATA TO APPLICATION ROLE app_role; GRANT SELECT ON ALL TABLES IN SCHEMA EXTERNAL_DATA.PUBLIC TO APPLICATION ROLE app_role;
```

Answer: ([SHOW ANSWER](#))

The correct answer is C. First, IMPORTED PRIVILEGES is required so that the Native App can see the database. Next, GRANT SELECT ON FUTURE TABLES IN SCHEMA EXTERNAL_DATA.PUBLIC limits the scope of future grants only to the PUBLIC schema. The other options do not provide future grants scoped to the Public schema, or they grant ALL access. Options B and D require that the database USAGE privilege be explicitly granted in addition to IMPORTED PRIVILEGES, which are granted automatically when a database is shared. Option E is incorrect because 'ALL TABLES' does not cover tables that will be created in the future.

NEW QUESTION: 148

You are developing a Snowflake Native App that requires specific Snowflake features and functionalities. To ensure compatibility and proper execution across different consumer environments, you need to define the minimum required Snowflake version. Which section in the manifest file is used to specify the minimum Snowflake version required by the application and what is the correct syntax?

- A. The 'dependencies' section. Syntax: `"dependencies": [{"snowflake_version": ">=7.0"}]`
- B. The 'application' section. Syntax: `"application": {"min_snowflake_version": "7.0"}`
- C. The 'configuration' section. Syntax: `"configuration": {"required_snowflake_version": "7.0"}`
- D. The 'setup' section. Syntax:

```
CREATE PROCEDURE setup() RETURNS STRING LANGUAGE SQL AS $$
  SET snowflake_version = CURRENT_VERSION();
  IF snowflake_version < '7.0' THEN
    RETURN 'Minimum Snowflake version requirement not met.'
  END IF;
$$.
```

- E. The 'api_version' section. Syntax: `"api_version":` along with in the 'applications' section.

Answer: (SHOW ANSWER)

The 'application' section within manifest file is used to specify minimum snowflake version required by Snowflake Native Application with the syntax `"application": {"min_snowflake_version": "7.0"}`. This attribute ensures that app installed is compatible with consumer account.

NEW QUESTION: 149

You've deployed a Snowflake Native Application to the Marketplace. A consumer reports an issue, and after investigating, you suspect a vulnerability related to SQL injection within a stored procedure. You want to implement best practices to mitigate this risk. Which of the following strategies are recommended to secure the stored procedure against SQL injection?

- A. Use parameterized queries or prepared statements whenever possible to prevent user input from being directly interpreted as SQL code.
- B. Implement strict input validation and sanitization to ensure that user input conforms to expected data types and patterns.
- C. Grant 'OWNERSHIP' privilege to the stored procedure to prevent unauthorized modifications.
- D. Encode all user input before inserting it into SQL queries to neutralize potentially harmful characters.
- E. Disable all external network access from within the stored procedure environment.

Answer: (SHOW ANSWER)

SQL injection vulnerabilities arise when user input is incorrectly incorporated into SQL queries. Parameterized queries (A) and input validation/sanitization (B) are crucial defenses. Parameterized queries treat input as data, not code, preventing injection. Input validation ensures only expected data reaches the query. Encoding user input (D) neutralizes harmful characters that could be exploited. Granting 'OWNERSHIP' (C) is for access control, not injection prevention. Disabling network access (E) might improve overall security but doesn't directly address SQL injection within the procedure.

NEW QUESTION: 150

You are developing a Snowflake Native Application that utilizes Snowpark Container Services (SPCS) to run a machine learning model within the customer's Snowflake environment. The application requires access to data stored in a secure stage within the customer's account. Considering security best practices and the limitations of SPCS in Native Apps, what is the most secure and recommended approach to grant the container access to the customer's data stage?

- A. Granting the application's service account the 'READ' privilege directly on the secure stage. This ensures the container can directly access the data.
- B. Creating a Snowflake UDF that reads data from the secure stage and exposes it through a secure view, which the container accesses using the application's service account.
- C. Utilizing a Snowflake external function that proxies the data access request to an external API gateway managed by the provider, which then fetches the data from the secure stage. This keeps the access credentials out of the customer's environment completely.
- D. Packaging the data within the application package during the build process. This eliminates the need for runtime access to the customer's data stage.
- E. Creating a Snowflake Stored Procedure with 'EXECUTE AS CALLER' rights and granting the application's service account execute access to the stored procedure. The stored procedure reads data from the secure stage and returns it to the container via a secure data sharing mechanism.

Answer: (SHOW ANSWER)

Options B and E are the most secure and recommended approaches. Option B uses a UDF and secure view to abstract the data access, limiting the container's direct access to the secure stage and leveraging Snowflake's security features. Option E uses a Stored Procedure executed as the caller to access the data on behalf of the application's service account. This allows the service account access to the stored procedure itself, but the access to the underlying stage is controlled by the owner of the stored procedure and not the application. Options A is less secure as it directly grants access to the stage. Option C, while secure in principle, overcomplicates the data access pattern and relies on external infrastructure. Option D is not feasible for large datasets and is generally not a scalable solution.

NEW QUESTION: 151

You are tasked with designing a Snowflake Native App that requires multiple roles with different levels of access to its functionalities. The app needs to define which roles can perform specific actions, such as viewing data, modifying data, or managing app settings. Which of the following methods are CORRECT ways to control role-based access within a Snowflake Native App? (Select all that apply)

- A.** Granting specific privileges to roles directly within the application's stored procedures and functions, checking the current role using and conditionally executing code based on the role.
- B.** Using the 'allowed_roleS' section in the manifest file to restrict access to certain functionalities based on the consumer's roles. This ensures that only authorized roles can interact with those components.
- C.** Creating separate versions of the application for each role, with each version containing only the functionalities that the role is allowed to access.
- D.** Defining application roles within the provider account and mapping them to consumer roles during the installation process. These application roles can then be used for fine-grained access control.
- E.** Defining roles within the setup script with specific privileges and instructing the consumer to grant their existing roles to these newly created roles.

Answer: (SHOW ANSWER)

Option A is correct because using within stored procedures and functions allows you to conditionally execute code based on the current role, providing fine-grained access control. Option D is also correct because defining application roles within the provider account and mapping them to consumer roles during installation allows for a structured and manageable approach to role-based access control within the app.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (378 Q&As Dumps, **35%OFF** Special Discount Code: **freecram**)

NEW QUESTION: 152

You are developing a Snowflake Native Application that processes sensitive customer data. You need to ensure that the application logs all relevant events for auditing and debugging purposes, while adhering to data privacy regulations. Which of the following strategies would BEST facilitate secure and compliant logging within the application provider's environment?

- A.** Log all events directly to a Snowflake table within the application's container, granting direct access to the consumer account for auditing.
- B.** Utilize Snowflake's Event Tables with appropriate masking policies to redact sensitive information before logging. Configure a data retention policy that complies with regulatory requirements. Define alerts based on specific event types to proactively address potential issues within the application.
- C.** Implement logging using standard Python logging libraries within the application code and direct the logs to stdout/stderr. Rely on the consumer to configure logging infrastructure within their account to capture the output.
- D.** Send logs to an external logging service via API calls, ensuring the service is SOC 2 compliant and offers end-to-end encryption. Consumers configure the service to receive logs and set up audit trails.
- E.** Encrypt all sensitive data within the logs using a key managed by the consumer before writing to a standard Snowflake table within the application container. Provide the consumer with the decryption key upon request.

Answer: (SHOW ANSWER)

Option B is the best approach. Snowflake's Event Tables provide a secure and managed environment for logging within the provider's account. Masking policies ensure data privacy by redacting sensitive information before logging. Retention policies help comply with regulations. Alerting allows for proactive issue resolution. Options A, C, D, and E present security or compliance risks.

NEW QUESTION: 153

You're developing a Snowflake Native App that interacts with a REST API. The API key needs to be securely stored and accessed within your application. Which of the following approaches provide the MOST secure and manageable way to handle the API key?

- A. Store the API key directly in the application code as a constant variable.
- B. Store the API key as an environment variable in the Snowflake environment.
- C. Use Snowflake Secrets to store the API key securely and grant access to the APPLICATION role. Then, access the secret within your Stored Procedure or UDF.
- D. Store the API key in a separate configuration file uploaded to a stage, encrypting the file with a symmetric key hardcoded in the application.
- E. Store the API key in a secure vault (e.g., HashiCorp Vault) and use an external function to retrieve the key at runtime. Implement appropriate authentication and authorization for the external function.

Answer: ([SHOW ANSWER](#))

Snowflake Secrets are designed for securely storing sensitive information like API keys. Granting access to the APPLICATION role ensures that only the application code can access the key. Storing keys directly in code or configuration files, even encrypted ones, is a security risk. Environment variables in Snowflake are not a secure storage mechanism for sensitive data. An external vault adds significant complexity to a Snowflake Native App.

NEW QUESTION: 154

You're developing a Snowflake Native Application that performs complex data transformations within the consumer's account. These transformations require substantial compute resources. You want to optimize resource utilization and minimize cost for your consumers.

Which strategies can you employ to achieve this goal? (Select TWO)

- A. Force consumers to use the largest possible virtual warehouse size to ensure transformations complete quickly.
- B. Implement dynamic scaling by programmatically adjusting the virtual warehouse size based on the workload using stored procedures and the function to cancel long running and inefficient tasks.
- C. Utilize Snowflake's auto-suspend and auto-resume features for virtual warehouses to minimize compute costs when the application is idle.
- D. Implement data skipping techniques to minimize the amount of data scanned during transformations. This involves clustering tables appropriately.
- E. Do not use transactions for data transformation pipelines as transactions introduce overhead.

Answer: ([SHOW ANSWER](#))

Snowflake's auto-suspend and auto-resume features (C) automatically suspend the virtual warehouse when it's idle and resume it when needed, minimizing compute costs. Data skipping techniques (D), such as clustering tables, reduce the amount of data scanned, which directly impacts compute cost. Option A is incorrect because forcing consumers to use the largest warehouse is not cost-effective. Option B is not a standard feature and might not be efficient. Option E is incorrect as transactions ensure data integrity, which is usually a paramount requirement.

NEW QUESTION: 155

You are developing a Snowflake Native Application. You need to grant specific privileges on a warehouse named 'APP WH' to the application role 'app_public' to allow application users to execute queries within that warehouse. Which of the following SQL statements is the MOST secure and appropriate way to achieve this, adhering to the principle of least privilege?

- A. `GRANT ALL PRIVILEGES ON WAREHOUSE APP_WH TO ROLE app_public;`
- B. `GRANT USAGE ON WAREHOUSE APP_WH TO ROLE app_public;`
- C. `GRANT OPERATE ON WAREHOUSE APP_WH TO ROLE app_public;`
- D. `GRANT MONITOR ON WAREHOUSE APP_WH TO ROLE app_public;`
- E. `GRANT OWNERSHIP ON WAREHOUSE APP_WH TO ROLE app_public;`

Answer: ([SHOW ANSWER](#))

The most secure and appropriate option is to grant the 'USAGE' privilege. Granting 'ALL PRIVILEGES' or 'OWNERSHIP' is overly permissive and violates the principle of least privilege. 'OPERATE' is not a valid privilege for warehouses. 'MONITOR' would only allow monitoring of the warehouse, not query execution. Only 'USAGE' allows the role to utilize the warehouse for executing queries.

NEW QUESTION: 156

You're developing a Snowflake Native Application with a front-end UI built using Streamlit, deployed as a UDF. This UI allows users to upload CSV files for processing. To adhere to security best practices and prevent unacceptable code practices in a Native App, what considerations should you prioritize in your development and deployment process specifically related to handling user-uploaded files ?

- A.** Implement strict file size limits and file type validation on both the client-side (Streamlit UI) and server-side (Snowflake UDF) to prevent denial-of-service attacks and malicious file uploads. Sanitize and validate the contents of the uploaded CSV file, escaping special characters and preventing SQL injection vulnerabilities if data from the CSV is used in SQL queries.
- B.** Store the uploaded CSV files directly within a Snowflake table without any validation to ensure high performance and minimal overhead.
- C.** Disable any logging or auditing of file upload activity to protect user privacy and avoid unnecessary data storage.
- D.** Assume that all uploaded files are safe and trustworthy, and directly process them without any security checks.
- E.** Allow user to execute scripts or upload executable file as part of CSV content

Answer: ([SHOW ANSWER](#))

Option A is the most secure approach. It emphasizes both client-side and server-side validation to prevent malicious uploads. Storing files directly without validation (B) is highly insecure. Disabling logging (C) hinders auditing and incident response. Assuming files are safe (D) is a critical security flaw. Option E is very dangerous from security perspective, it should not be allowed.

NEW QUESTION: 157

You have a Snowflake Native Application that utilizes a managed table. After upgrading your application to version 2.0, some users report that they are missing data in the managed table. After investigation, you realize version 2.0 changed the schema of the managed table and also did not properly migrate all the data from old format into new format. Which of the following steps should be taken to address the missing data after upgrading a Snowflake Native Application?

- A.** Create a new version 2.1 that includes a data migration script to transfer data from the old schema (as it existed in version 1.0) to the new schema in version 2.0. Use release directives to upgrade affected users to version 2.1 after staging.
- B.** Rollback all users to version 1.0 and then perform the schema migration directly on the consumer's account using custom SQL scripts.
- C.** Manually insert the missing data into the managed table in each consumer's account, contacting affected users to coordinate the data entry.
- D.** Create a new version 2.1. Within version 2.1's setup script, dynamically detect if the application is an upgrade from 1.0 and, if so, execute the data migration script to transfer data from the old schema to the new schema.
- E.** Since managed tables are fully managed by Snowflake, contact Snowflake support to initiate a data recovery process in each affected consumer account.

Answer: ([SHOW ANSWER](#))

Option D is the correct approach. By dynamically detecting the upgrade and running a data migration script, the application ensures data integrity during the upgrade process. The data migration script will convert data from old format to new format. Option A is almost correct, however we don't assume the user is at 1.0. The application may be updated from earlier versions as well. Option B is not feasible due to operational overhead of doing it for each consumers and requires high privilege on consumer side which is not allowed for native apps. Option C is not practical or scalable. Managed tables are managed by your application code within the package, so Snowflake support won't perform data manipulation (Option E).

NEW QUESTION: 158

You are developing a Snowflake Native App that performs data transformations. The core logic is implemented in a series of stored procedures. You need to implement versioning for your app to allow for future updates and rollbacks. Which of the following steps are REQUIRED to properly implement versioning for a Snowflake Native App?

- A.** Increment the 'version' attribute in the 'MANIFEST.yml' file for each new release.
- B.** Create a separate branch in your version control system for each version of the app.
- C.** Update the 'version' attribute in the 'setup.sqr' file, if one exists.
- D.** Use the 'ALTER APPLICATION' command to update the app's version in the consumer's account.
- E.** Tag each release in your version control system with the corresponding version number.

Answer: ([SHOW ANSWER](#))

The 'version' attribute in the 'MANIFEST.yml' file is the core mechanism Snowflake uses to identify different versions of your Native App. Incrementing this value for each release is essential for proper versioning. The other options, while potentially part of a good software development lifecycle, are not directly related to how Snowflake handles Native App versions. ALTER APPLICATION is used to update the PATCH version, not the main version declared in MANIFEST.yml. No ALTER APPLICATION command is used in provider account

NEW QUESTION: 159

You are designing a Snowflake Native Application that performs sentiment analysis on customer reviews. The application needs to store intermediate results and provide aggregated sentiment scores to the consumer. You are considering using either temporary tables or regular tables within your application package. Which of the following considerations are most important when deciding between these two options?

- A. Temporary tables are automatically dropped at the end of each session, ensuring that no sensitive data is persisted longer than necessary. This is crucial for maintaining data privacy and compliance.
- B. Temporary tables are visible to all roles within the application package, simplifying data access and management for different components of the application.
- C. Regular tables are persisted across sessions and can be queried by the consumer after the application has completed its processing. This allows the consumer to analyze the intermediate results and gain further insights.
- D. Regular tables require specifying a data retention period, which adds complexity to the application's setup script and can potentially lead to data loss if not configured correctly.
- E. The data retention period for regular tables can be used to implement data versioning and auditing within the application package.

Answer: ([SHOW ANSWER](#))

Options A, C, and E are correct. Temporary tables ensure data privacy by being session-bound. Regular tables allow consumers access to results and enable data versioning. Option B is incorrect because temporary tables are only available to the session that created them. Option D is partially correct, but the benefit of data versioning outweighs the cost of implementing retention period.

NEW QUESTION: 160

You are developing a Snowflake Native Application that leverages Snowflake's Snowpark API for data processing. The application performs a series of complex transformations on a DataFrame. You need to optimize the application's performance and minimize resource consumption. Which of the following strategies would be MOST effective in achieving this goal?

- A. Use the `collect()` method frequently to materialize intermediate DataFrames and inspect the data.
- B. Rely on Snowpark's query optimizer to automatically optimize the entire data processing pipeline without any manual intervention.
- C. Employ lazy evaluation by chaining together transformations on the DataFrame and only triggering execution when the final result is needed.
- D. Explicitly specify the execution order of the transformations using `'cache()'` on each intermediate DataFrame to force immediate materialization.
- E. Break down the DataFrame processing into smaller steps and store the intermediate results in temporary tables within the application's sandbox database.

Answer: ([SHOW ANSWER](#))

Lazy evaluation (C) is the most effective strategy. By chaining transformations and only executing when the final result is needed, Snowpark's query optimizer can optimize the entire pipeline, reducing resource consumption. `'collect()'` (A) materializes DataFrames unnecessarily, hindering optimization. While Snowpark does optimize queries (B), relying solely on it is insufficient. Explicit materialization (D) defeats the purpose of lazy evaluation. Storing intermediate results in temporary tables (E) adds overhead and complexity.

NEW QUESTION: 161

You are preparing to deploy a Snowflake Native Application to the Snowflake Marketplace. You have implemented versioning using the `'VERSION'` clause in your setup script. Your application relies on a specific shared data source that is only available in the AWS us-west-2 region. To ensure that consumers in other regions can still install and use your application, you decide to offer different versions with region-specific logic. Which of the following approaches is the MOST effective and recommended to manage region-specific configurations and data sharing across multiple versions of your Snowflake Native Application?

- A. Create separate application packages for each region, containing region-specific configurations and data sharing setups. This ensures consumers only install the version relevant to their region.
- B. Implement conditional logic within the application's stored procedures to check the current region and adjust data sharing and execution accordingly using Package all configurations within a single version.
- C. Use the `'PATCH'` clause during application deployment to selectively modify the configuration based on the target region. Deploy the same application package to all regions.
- D. Implement a build process that generates region-specific setup scripts based on environment variables using a CI/CD pipeline. Each version of the application will then contain the right setup script based on the target region.

E. Use separate schemas within the application package for each region, and configure application roles with permissions to only access schemas relevant to their region. No changes to code logic are required.

Answer: ([SHOW ANSWER](#))

The most effective approach is to utilize a CI/CD pipeline to generate region-specific setup scripts. This allows you to manage region-specific configurations declaratively within your code repository and ensures a consistent and repeatable deployment process across all regions. Creating separate application packages (Option A) becomes difficult to manage and update. Conditional logic (Option B) can become complex and harder to maintain. PATCH (Option C) is not intended for such fundamental configuration differences. Using different schemas (Option E) does not address where the data resides, it just provides access control.

Valid NAS-C01 Dumps shared by EduDump.com for Helping Passing NAS-C01 Exam! EduDump.com now offer the **newest NAS-C01 exam dumps**, the EduDump.com NAS-C01 exam **questions have been updated** and **answers have been corrected** get the **newest** EduDump.com NAS-C01 dumps with Test Engine here: <https://www.edudump.com/exams/Snowflake/NAS-C01/premium/> (**378 Q&As** Dumps, **35%OFF** Special Discount Code: **freecram**)