Databricks Data Engineer Associate Master Cheat Sheet

Section 1: Databricks Lakehouse Platform

Relationship between Data Lakehouse and Data Warehouse:

- **Data Lake:** Stores all raw data in its original format, regardless of structure or schema. Useful for exploratory analysis and flexible data exploration. However, data quality and consistency can be challenging.
- **Data Warehouse:** Stores structured and curated data, optimized for querying and analysis. Ensures data quality and consistency but may not accommodate all data types or flexible exploration.
- **Data Lakehouse:** Combines the benefits of both: stores raw data like a data lake while allowing structured and curated data like a data warehouse. Enables flexible exploration with improved data quality and consistency.

Data Quality Improvement in Data Lakehouse:

- Data lakehouses employ data cleansing, transformation, and schema enforcement techniques to improve data quality compared to raw data lakes.
- This allows for more reliable analysis and reduces the risk of errors.

Silver vs. Gold Tables:

- **Bronze Table:** Stores the first ingestion of raw data, often unprocessed and potentially containing errors. Used for initial data landing and exploration.
- **Silver Table:** Contains data that has undergone some level of processing and validation, improving quality compared to bronze tables. Used for broader data analysis and reporting.
- **Gold Table:** Represents the highest level of data quality, meeting all business requirements for accuracy, consistency, and completeness. Used for critical decision-making and downstream applications.

Databricks Platform Architecture:

- **Data Plane:** Handles data processing tasks, including clusters, notebooks, jobs, and libraries. Runs in the customer's cloud account.
- **Control Plane:** Manages user accounts, permissions, clusters, and workspace configurations. Resides in a separate, secure environment managed by Databricks.

All-Purpose vs. Job Clusters:

- **All-Purpose Clusters:** Long-running clusters suitable for interactive development, exploration, and various workloads. Offer flexibility but can be less cost-effective for specific jobs.

- **Job Clusters:** Short-lived clusters created and terminated automatically for running specific jobs. More cost-efficient for focused tasks but less suitable for interactive work.

Databricks Runtime Versioning:

- Databricks Runtime (DBR) manages cluster software versions.
- Users can choose from pre-built configurations (e.g., DBR 10.5) or customize them with specific libraries and dependencies.

Filtering Clusters:

- Users can filter clusters based on various criteria like:
    - **Cluster mode:** all-purpose, job, or instance pool
    - **Active/terminated status**
    - **Creator**
    - **Tags** assigned to the cluster

Terminating and Impact of Termination:

- **Terminating a cluster:**
    - Stops all running jobs and applications on the cluster.
    - Releases resources associated with the cluster, reducing costs.
    - Can be done through the **Clusters** UI or programmatically (API or CLI).
- **Impact of termination:**
    - Any unfinished work is lost.
    - Cluster configurations and data are preserved unless explicitly deleted.
    - Restarting the cluster incurs new resource costs.

Restarting Clusters:

- **Scenarios for restarting:**
    - Cluster encounters unexpected errors or issues.
    - Need to update cluster configuration (e.g., libraries, versions).
    - Desire to refresh cluster resources for improved performance.
- **Restarting does not:**
    - Fix underlying code issues causing errors.
    - Recover lost data from terminated jobs.

Using Multiple Languages in Notebooks:

- Databricks notebooks support magic commands to use different languages:
    - %python: Execute Python code.

- %scala: Execute Scala code.
- %r: Execute R code.
- You can switch between languages within the same notebook.

Running Notebooks from Other Notebooks:

- Use the dbutils.notebook.run function:

Python

dbutils.notebook.run("/path/to/other/notebook", arguments={"arg1": value1, "arg2": value2})

Use code with caution.

content_copy

- Pass arguments to the called notebook for dynamic execution.

Sharing Notebooks:

- Share notebooks with other users by:
  - Granting access permissions in the **Workspace** settings.
  - Publishing notebooks to the **Community Notebooks** section.

Databricks Repos and CI/CD:

- Databricks Repos enables version control for notebooks, clusters, and libraries.
- Integrates with CI/CD tools like Jenkins or GitLab to automate workflows:
  - Code commits trigger automated testing and deployment of notebooks.
  - Version control ensures rollback capability and collaboration.

Git Operations in Databricks Repos:

- Basic Git operations available:
  - Clone, commit, push, pull, branch, merge.
  - View revision history and collaborate with others.

Limitations of Notebook Version Control:

- Individual cell history not tracked in notebooks.
- Limited branching and merging capabilities compared to dedicated Git repositories.

Section 2: ELT with Apache Spark

Extracting Data:

- **Single File:** Use spark.read.format("format").load("path/to/file") where "format" is the file type (e.g., csv, json, parquet).

- **Directory of Files:** Use spark.read.format("format").load("path/to/directory") or read all files recursively with spark.read.format("format").option("recursiveFileLookup", true).load("path/to/directory").

Data Type Prefix:

Spark infers the data type based on the file extension or header information. The prefix after FROM can be:

- csv - Comma-separated values
- json - JSON format
- parquet - Parquet columnar format

Views, Temporary Views, and CTEs:

- **View:** Defined with CREATE VIEW view_name AS SELECT ... FROM .... Accessible like a table but not stored physically.
- **Temporary View:** Defined with CREATE TEMPORARY VIEW view_name AS SELECT ... FROM .... Exists only within the current Spark session.
- **CTE (Common Table Expression):** Defined within a query with WITH cte_name AS (SELECT ... FROM ...) SELECT ... FROM cte_name. Useful for complex subqueries.

External Tables vs. Delta Lake Tables:

- **External Tables:** Point to data stored outside of Delta Lake (e.g., CSV, Parquet). Not managed by Delta Lake.
- **Delta Lake Tables:** Managed by Delta Lake, providing features like versioning, ACID transactions, and schema enforcement.

Creating Tables:

- **JDBC Connection:** Use spark.read.jdbc("url", "table_name", connectionProperties). Requires JDBC driver and connection details.
- **CSV File:** Use spark.read.format("csv").option("header", true).load("path/to/file"). Optionally specify schema or infer from header.

Counting Functions:

- **count_if(condition):** Counts rows where the condition is true.
- **count(where x is null):** Counts rows where column x is null.
- **count(row):** Counts all rows, excluding null values.

Deduplication:

- **Existing Delta Lake Table:** Use dropDuplicates method on the DataFrame.
- **New Table:** Use select distinct or dropDuplicates while creating the new table.
- **Specific Columns:** Use dropDuplicates with a list of column names.

Primary Key Validation:

Use show create table table_name to view table schema and identify the primary key. You can then write queries to check for duplicate values in the primary key columns.

Validate Unique Values:

- **Spark SQL:** Use COUNT DISTINCT or GROUP BY with aggregation functions like COUNT or MAX to identify duplicate values.
- **DataFrame API:** Utilize df.groupBy("field1").agg(count("field2")).filter("count > 1") to find fields with more than one unique value in another field.

Validate Missing Values:

- **Spark SQL:** Use ISNULL or COALESCE functions to check for null values.
- **DataFrame API:** Apply df.filter("field IS NULL") to identify rows with missing values in a specific field.

Cast Column to Timestamp:

- **Spark SQL:** Use to_timestamp function with appropriate format specifier, e.g., df.withColumn("timestamp_col", to_timestamp("string_col", "yyyy-MM-dd HH:mm:ss")).
- **DataFrame API:** Utilize df.withColumn("timestamp_col", $"string_col".cast("timestamp")) for simpler conversion.

Extract Calendar Data:

- **Spark SQL:** Leverage functions like year, month, day, and weekofyear to extract specific calendar components from timestamps.
- **DataFrame API:** Utilize similar functions on the timestamp column, e.g., df.withColumn("year", $"timestamp_col".year).

Extract Specific Patterns:

- **Spark SQL:** Use regular expressions with regexp_extract function to match and extract desired patterns from strings.
- **DataFrame API:** Apply df.withColumn("extracted_pattern", regexp_extract($"string_col", "pattern", 1)) to extract the first match of the pattern.

Utilize Dot Syntax:

- Access nested data structures within columns using dot notation, e.g., df.select("data.field1", "data.field2").

Benefits of Array Functions:

- **Manipulation:** map, filter, explode, and other functions enable efficient transformations on arrays of data.
- **Aggregation:** Functions like sum, avg, and count can be applied to arrays for aggregate calculations.

- **Improved Performance:** Array functions can optimize processing compared to working with individual elements.

Parse JSON strings into structs:

- JSON is a popular data format used to represent semi-structured data.
- Spark SQL provides functions like from_json and get_json_object to parse JSON strings into structured data (structs).
- This allows you to extract specific values from the JSON and use them in your queries.

Identify which result will be returned based on a join query:

- Joins are used to combine data from multiple datasets based on a shared column.
- Different join types (e.g., inner join, left join, right join) determine which rows are included in the resulting dataset.
- Understanding join types is crucial for accurately combining data from different sources.

Identify a scenario to use the explode function versus the flatten function:

- Both explode and flatten functions are used to transform nested data structures.
- explode expands an array into separate rows, one for each element in the array.
- flatten can handle more complex nested structures, flattening them into a single level.
- Choosing the right function depends on the specific structure of your data and desired outcome.

Identify the PIVOT clause as a way to convert data from wide format to long format:

- Wide format data stores multiple attributes in separate columns.
- Long format stores attributes as rows, with one attribute per column and another column for the corresponding value.
- The PIVOT clause allows you to transform data from wide format to long format, making it easier to analyze and visualize.

Define a SQL UDF (User-Defined Function):

- UDFs allow you to extend Spark SQL with custom logic that can be used within your queries.
- You can define UDFs in various languages like Python, Scala, or SQL.
- UDFs can be helpful for performing complex transformations or calculations not directly supported by Spark SQL functions.

Identify the location of a function:

- Spark SQL functions can be built-in or user-defined.
- Built-in functions come with Spark SQL and are available by default.
- User-defined functions are defined by you and need to be registered in the appropriate catalog (e.g., session catalog, global catalog) before use.

Describe the security model for sharing SQL UDFs:

- Sharing UDFs requires careful consideration of security implications.
- Spark SQL provides different access control mechanisms to regulate who can use and modify UDFs.
- Understanding these mechanisms ensures responsible and secure sharing of custom functions.

Use CASE/WHEN in SQL code:

- CASE/WHEN is a conditional expression used to perform different actions based on specific conditions.
- It allows you to implement logic within your SQL queries, making them more flexible and powerful.

Leverage CASE/WHEN for custom control flow:

- By combining CASE/WHEN with other SQL functions and operators, you can achieve complex control flow within your queries.
- This allows you to manipulate data based on various conditions and perform customized transformations.

Section 3: Incremental Data Processing

Identify where Delta Lake provides ACID transactions:

Delta Lake tables support ACID transactions at the **data file level**. This means individual data files within a Delta table can be written atomically (all or nothing), consistently (data remains consistent across updates), isolated (multiple writes don't interfere), and durably (data persists even in case of failures).

Identify the benefits of ACID transactions:

ACID transactions offer several benefits:

- **Data consistency:** Ensures data updates are complete and valid, preventing partial or inconsistent writes.
- **Data integrity:** Protects data from corruption during concurrent writes or failures.
- **Rollback capability:** Allows reverting to a previous state in case of errors.
- **Improved reliability:** Guarantees data consistency across various operations.

Identify whether a transaction is ACID-compliant:

To determine if a transaction is ACID-compliant, check if it fulfills all four properties:

- **Atomicity:** The entire transaction succeeds or fails as a whole.
- **Consistency:** The transaction brings the database from one valid state to another.
- **Isolation:** Concurrent transactions don't interfere with each other's data.

- **Durability:** Once committed, changes persist even in case of failures.

Compare and contrast data and metadata:

- **Data:** The actual information stored in a table, like user IDs, product names, etc.
- **Metadata:** Information about the data itself, such as schema definition, creation time, access permissions, etc.

Compare and contrast managed and external tables:

- **Managed tables:** Stored entirely within the Delta Lake storage layer, including data and metadata. Databricks manages the data files and directory structure.
- **External tables:** Reference data stored outside of Delta Lake, like in cloud storage (e.g., S3, ADLS). Databricks uses the provided schema to access the data but doesn't manage the underlying files.

Identify a scenario to use an external table:

Use external tables when:

- Data already resides in another storage system and doesn't need to be managed by Databricks.
- You want to leverage existing data pipelines or tools that work with the external storage.
- Reducing storage costs by keeping data in a separate location.

Create a managed table:

You can create a managed table using various methods:

- **Spark SQL:** CREATE TABLE statement with Delta Lake format specified.
- **Databricks UI:** Use the table creation interface and select Delta Lake as the storage format.
- **Python API:** Utilize the spark.createDataFrame function and specify Delta format during saving.

Identify the location of a table:

The table location refers to the physical storage path where Delta Lake stores the data files and metadata. You can find this information using:

- **Spark SQL:** DESCRIBE EXTENDED table_name command.
- **Databricks UI:** View the table properties in the UI.
- **Python API:** Use the table.location property of the DataFrame associated with the table.

Inspect the directory structure of Delta Lake files:

Delta Lake stores data in a versioned format, with each version represented by a directory containing data files and metadata files. You can explore this structure using:

- **Databricks File System (DBFS):** Browse the table location in DBFS to see the directory structure.

- **Spark SQL commands:** Use commands like SHOW FILES IN LOCATION to list files within the table directory.

Identify who has written previous versions of Delta Lake table:

Delta Lake tracks information about who wrote each version of the table data. You can access this information using:

- **Spark SQL:** DESCRIBE HISTORY table_name command shows details like timestamps and usernames for each version.
- **Databricks UI:** The table history section in the UI displays information about previous versions and their creators.

Review a history of table transactions:

Delta Lake tables track all changes made to the data, allowing you to view a history of inserts, updates, and deletes. This enables features like rollbacks and querying specific versions.

Roll back a table to a previous version:

If you encounter issues with your data, you can revert a Delta Lake table to a previous point in time using the ALTER TABLE ... SET TIMESTAMP AS <timestamp> syntax. This restores the table's state based on the specified timestamp.

Identify that a table can be rolled back to a previous version:

Understanding Delta Lake's versioning capabilities is crucial. Unlike traditional tables, Delta Lake allows reverting to previous states, providing a safety net for data manipulation mistakes.

Query a specific version of a table:

By leveraging the versioning feature, you can query a specific point-in-time snapshot of the data using the AS OF TIMESTAMP <timestamp> clause in your SQL queries. This enables historical analysis or rollbacks.

Identify why Zordering is beneficial to Delta Lake tables:

Zordering organizes data files in the table based on their partition and sort columns. This significantly improves query performance by allowing efficient data skipping during scans.

Identify how vacuum commits deletes:

Vacuum commits are Delta Lake operations that reclaim storage space by physically removing deleted data files from the table storage. This helps maintain efficient storage utilization.

Identify the kind of files Optimize compacts:

Optimize compacts rewrite existing data files in a Delta Lake table, combining smaller files into larger ones and potentially reorganizing data for better performance. This improves query efficiency by reducing the number of files to scan.

Identify CTAS as a solution:

CTAS (CREATE TABLE AS SELECT) is a powerful technique for creating new Delta Lake tables from existing data sources. It allows efficient data transformation and loading in a single operation.

Create a generated column:

Generated columns are virtual columns automatically computed based on an expression defined during table creation or modification. They don't store actual data but are calculated on-the-fly during queries, improving query performance and reducing data redundancy.

Add a table comment:

Adding comments to Delta Lake tables provides valuable metadata for documentation purposes. You can use the ALTER TABLE ... SET COMMENT <comment> syntax to add or modify table comments.

CREATE OR REPLACE TABLE and INSERT OVERWRITE:

- **CREATE OR REPLACE TABLE:**
    - Creates a new table if it doesn't exist or overwrites the existing one with new data.
    - Used for initial data loading or complete table updates.
- **INSERT OVERWRITE:**
    - Inserts new data into a table, overwriting any existing data with the same partition keys.
    - Faster than CREATE OR REPLACE TABLE for large datasets.

Comparison and Contrast:

Feature	CREATE OR REPLACE TABLE	INSERT OVERWRITE
Existing table	Creates new or replaces existing	Overwrites existing only
Data deletion	Deletes all existing data	Deletes only data with matching partition keys
Speed	Slower for large datasets	Faster for large datasets
Use case	Initial data load, complete table updates	Incremental updates, partial data overwrites

drive_spreadsheetExport to Sheets

MERGE for Deduplication:

- **MERGE:** Combines data insertion, update, and deletion based on specified conditions.
- Can be used for deduplication by inserting unique records and updating or deleting duplicates based on a unique identifier.

Identifying MERGE for Deduplication:

Look for scenarios where you need to:

- **Insert new data** while **preventing duplicates** based on specific criteria.
- **Update existing data** with new values while **handling duplicates**.
- **Delete duplicate records** based on a unique identifier.

Benefits of MERGE:

- **Efficient deduplication:** Handles data insertion, update, and deletion in a single operation.
- **Flexible conditions:** Supports various conditions for identifying and handling duplicates.
- **Upsert functionality:** Inserts new data and updates existing records efficiently.

COPY INTO Not Deduplicating Data:

- **COPY INTO:** Primarily used for bulk data loading from external sources.
- It doesn't have built-in deduplication functionality.
- If duplicates exist in the source data, they will be copied into the target table as well.

Identify a scenario in which COPY INTO should be used:

- Use COPY INTO when you need to **bulk load large datasets** into Delta Lake tables from various external sources like CSV, JSON, Parquet files.
- It's efficient for large datasets due to parallelization and optimized data ingestion.

Use COPY INTO to insert data:

- Define the source location (e.g., file path), format (e.g., CSV), and target Delta Lake table.
- Use options like schema to specify data schema, partitionBy for partitioning, and overwrite for behavior (append or overwrite).

Identify the components necessary to create a new DLT pipeline:

- **Notebook:** Contains code to define data transformations and load data into Delta Lake.
- **Job:** Executes the notebook code on a cluster.
- **Trigger:** Schedules the job execution based on time or events.
- **Cluster:** Provides compute resources for job execution.
- **Delta Lake table:** Stores the processed data.

Identify the purpose of the target and of the notebook libraries in creating a pipeline:

- **Target:** Defines where the processed data is stored (Delta Lake table)
- **Notebook libraries:** Provide functions for data manipulation, transformation, and loading into Delta Lake (e.g., Spark SQL, Databricks Runtime libraries)

Compare and contrast triggered and continuous pipelines in terms of cost and latency:

- **Triggered pipelines:**
    - Run at specific times or events (e.g., daily, hourly)
    - Lower cost as they run less frequently
    - Introduce latency between trigger and data availability
- **Continuous pipelines:**
    - Run continuously as data arrives (e.g., using Auto Loader)
    - Higher cost due to constant resource usage
    - Minimal latency, data is available almost instantly

Identify which source location is utilizing Auto Loader:

- Look for configurations like format option set to auto or presence of autoloader directory in the source path.

Identify a scenario in which Auto Loader is beneficial:

- Use Auto Loader when you need **near real-time data ingestion** from continuously changing sources like log files, streaming data feeds.
- It automatically detects new files and loads them into Delta Lake without manual intervention.

Auto Loader Inferring All Data as String:

- Auto Loader automatically infers schema for JSON data but may interpret everything as String if it cannot confidently identify specific data types.
- Look for complex data structures, nested objects, or missing type information in your JSON source.
- Use data type hints or manually define the schema to ensure accurate data types.

Default Behavior of Constraint Violation:

- Databricks enforces constraints by default, meaning invalid data will cause row rejection.
- This behavior can be changed with constraint options like IGNORE or UPDATE NULL.

Impact of Constraint Violation Options:

- ON VIOLATION DROP ROW: Deletes the entire row containing the violation, potentially impacting downstream data.
- ON VIOLATION FAIL UPDATE: Stops the update operation on the row and returns an error, preserving existing data.

Change Data Capture & APPLY CHANGES INTO:

- Change Data Capture (CDC) tracks modifications in a table and replicates changes to another table.
- APPLY CHANGES INTO command defines which changes (inserts, updates, deletes) are replicated and the target table.

Querying Events Log:

- Events log stores information about Databricks operations like job runs, cluster events, and user actions.
- Use Spark SQL or Databricks SQL to query the events log for metrics, audit logs, and lineage analysis.

Troubleshooting DLT Syntax:

- Data Lifecycle Management (DLT) pipelines automate data management tasks like archiving and deletion.
- Identify error sources:
    - Notebook errors can be found in the notebook associated with the failed DLT operation.
    - LIVE keyword is required for reading data from Delta tables in the CREATE statement.
    - STREAM keyword is used in the FROM clause when reading data from a streaming source.

Section 4: Production Pipelines

Benefits of Multiple Tasks in Jobs:

- **Modularization:** Break down complex pipelines into smaller, reusable tasks for clarity and maintainability.
- **Parallelism:** Run independent tasks concurrently to improve processing speed.
- **Orchestration:** Define dependencies between tasks to ensure data flows in the desired order.
- **Error Handling:** Isolate failures to specific tasks, making debugging easier.

Setting Up Predecessor Tasks:

- Use the "depends_on" parameter in your task definition to specify a predecessor task.
- The predecessor must finish successfully before the current task starts.
- This ensures data is available for subsequent tasks, preventing errors.

Scenarios for Predecessor Tasks:

- Transforming data in one task before loading it into another.
- Running data quality checks before downstream analysis.
- Waiting for external data sources to become available.

Reviewing Task Execution History:

- View past task runs in the Jobs UI, including start/end times, logs, and outputs.
- Analyze successful and failed runs to identify issues and optimize performance.

CRON Scheduling:

- Schedule tasks to run periodically using CRON expressions.
- Examples: hourly runs ("0 * * * *"), daily runs ("0 0 * * *").
- Useful for automating regular data processing and pipeline execution.

Debugging Failed Tasks:

- Analyze task logs for error messages and stack traces.
- Use debugging tools like Spark UI or Jupyter Notebooks to inspect data and code.
- Identify the root cause of failure and implement fixes.

Retry Policies:

- Configure tasks to automatically retry upon failure within a specified limit.
- Useful for handling transient errors like network issues or resource limitations.
- Configure retry intervals and backoff strategies for optimal performance.

Creating Alerts for Failed Tasks:

- Set up alerts to be notified via email or other channels when tasks fail.
- Helps monitor pipeline health and proactively address issues.
- Configure alerts based on specific failure conditions or severity levels.

Email Alerts:

- Databricks allows sending email alerts triggered by task failures or other events.
- Configure recipient email addresses and message content.
- Ensure timely notifications for critical failures.

Section 5: Data Governance

Identify one of the four areas of data governance.

Data governance has four main areas:

- **Data Quality:** Ensures data accuracy, consistency, and completeness. Includes data profiling, cleansing, and validation.
- **Data Security:** Protects data from unauthorized access, use, disclosure, or modification. Involves access control, encryption, and logging.

- **Data Privacy:** Complies with regulations and policies regarding personal data collection, use, and storage. Includes data masking, anonymization, and deletion.
- **Data Lineage:** Tracks data movement and transformation throughout processing pipelines. Enables auditing, debugging, and impact analysis.

Compare and contrast metastores and catalogs.

Metastores:

- Store metadata about data (e.g., schema, location, lineage).
- Often specific to a data platform (e.g., Hive Metastore).
- May not have advanced features like data discovery or lineage visualization.

Catalogs:

- Provide a unified view of metadata across various data sources and platforms.
- Offer data discovery, lineage visualization, and governance features.
- Example: Unity Catalog in Databricks.

Identify Unity Catalog securables.

Unity Catalog securables are objects you can grant access to or revoke access from:

- Databases
- Tables
- Views
- Functions
- Locations (e.g., S3 buckets)

Define a service principal.

A service principal is an identity used by applications or services to access resources in Azure Active Directory (AAD). It allows applications to act on behalf of a user without needing a human password.

Identify the cluster security modes compatible with Unity Catalog.

- **User-managed ACLs:** Users manage access directly on Unity Catalog objects.
- **Service principal ACLs:** Service principals manage access using their own permissions.
- **External Identity Management (EIM) integration:** Integrates with external identity providers like AAD for access control.

Create a UC-enabled all-purpose cluster.

This involves configuring your cluster to use Unity Catalog for metadata management. You can do this through the Databricks web UI, CLI, or API. Refer to Databricks documentation for specific instructions.

Create a DBSQL warehouse.

A DBSQL warehouse is a serverless SQL query engine in Databricks. You can create one through the web UI or CLI, specifying details like size and configuration.

Identifying how to query a three-layer namespace:

- Databricks uses a hierarchical namespace for data objects: catalogs, databases, and tables.
- You can query objects across different layers using their full path, e.g., catalog_name.database_name.table_name.
- Understand the structure and purpose of each layer for efficient navigation and access control.

Implementing data object access control:

- Databricks supports various mechanisms for granular access control to data objects:
    - **ACLs (Access Control Lists):** Grant specific permissions (read, write, execute) to users and groups.
    - **RBAC (Role-Based Access Control):** Assign predefined roles with specific permissions to users and groups.
    - **External Identity Providers:** Integrate with external directories (e.g., Active Directory) for centralized authorization.
- Implementing appropriate access control ensures data security and compliance.

Identifying colocating metastores with a workspace as best practice:

- Databricks metastores store data object metadata (schema, location, ownership).
- Colocating a metastore with a workspace keeps metadata locally and improves performance.
- Consider using managed metastores for scalability and easier management.

Identifying using service principals for connections as best practice:

- Service principals are identities used by applications to connect to resources securely.
- Using service principals instead of user credentials avoids storing sensitive information in code and improves security.
- Databricks supports service principals for various integrations, like accessing data sources or cloud storage.

Identifying the segregation of business units across catalogs as best practice:

- Databricks allows creating multiple catalogs to organize data objects by business unit or function.
- Segregation provides better data isolation, access control, and lineage tracking.
- It simplifies data governance by managing access and policies for each business unit separately.

Disclaimer: All data and information provided on this site is for informational purposes only. This site makes no representations as to accuracy, completeness, correctness, suitability, or validity of any information on this site & will not be liable for any errors, omissions, or delays in this information or any losses, injuries, or damages arising from its display or use. All information is provided on an as-is basis.