DATABRICKS DOCUMENTATION

Associate Data Engineering

Describe the relationship between the data lakehouse and the data warehouse.

Data Lakehouse

A data lakehouse is a data management system that combines the benefits of data lakes and data warehouses. This article describes the lakehouse architectural pattern and what you can do with it on Databricks.

A data lakehouse provides scalable storage and processing capabilities for modern organizations that want to avoid isolated systems for processing different workloads, like machine learning (ML) and business intelligence (BI). A data lakehouse can help establish a single source of truth, eliminate redundant costs, and ensure data freshness.

Data lakehouses often use a data design pattern that incrementally improves, enriches, and refines data as it moves through layers of staging and transformation. Each layer of the lakehouse can include one or more layers. This pattern is frequently referred to as a medallion architecture.

Data Warehouse

Data warehousing refers to collecting and storing data from multiple sources so it can be quickly accessed for business insights and reporting. This article contains key concepts for building a data warehouse in your data lakehouse.

The lakehouse architecture and Databricks SQL bring cloud data warehousing capabilities to your data lakes. Using familiar data structures, relations, and management tools, you can model a highly-performant, cost-effective data warehouse that runs directly on your data lake.

As with a traditional data warehouse, you model data according to business requirements and then serve it to your end users for analytics and reports. Unlike a traditional data warehouse, you can avoid siloing your business analytics data or creating redundant copies that quickly become stale.

Building a data warehouse inside your lakehouse lets you bring all your data into a single system and lets you take advantage of features such as Unity Catalog and Delta Lake.

DL vs DW

Data warehouses have powered business intelligence (BI) decisions for about 30 years, having evolved as a set of design guidelines for systems controlling the flow of data.
Enterprise data warehouses optimize queries for BI reports, but can take minutes or even hours to generate results. Designed for data that is unlikely to change with high frequency, data warehouses seek to prevent

conflicts between concurrently running queries. Many data warehouses rely on proprietary formats, which often limit support for machine learning. Data warehousing on Databricks leverages the capabilities of a Databricks lakehouse and Databricks SQL.

Powered by technological advances in data storage and driven by exponential increases in the types and volume of data, data lakes have come into widespread use over the last decade. Data lakes store and process data cheaply and efficiently. Data lakes are often defined in opposition to data warehouses: A data warehouse delivers clean, structured data for BI analytics, while a data lake permanently and cheaply stores data of any nature in any format. Many organizations use data lakes for data science and machine learning, but not for BI reporting due to its unvalidated nature.

The data lakehouse combines the benefits of data lakes and data warehouses and provides:

- Open, direct access to data stored in standard data formats.
- Indexing protocols optimized for machine learning and data science.
- Low query latency and high reliability for BI and advanced analytics.

By combining an optimized metadata layer with validated data stored in standard formats in cloud object storage, the data lakehouse allows data scientists and ML engineers to build models from the same data-driven BI reports.

Compare and contrast silver and gold tables, which workloads will use a bronze table as a source, which workloads will use a gold table as a source.

Bronze layer

Data can enter your lakehouse in any format and through any combination of batch or steaming transactions. The bronze layer provides the landing space for all of your raw data in its original format. That data is converted to Delta tables.

Silver layer

The silver layer brings the data from different sources together. For the part of the business that focuses on data science and machine learning applications, this is where you start to curate meaningful data assets. This process is often marked by a focus on speed and agility.

The silver layer is also where you can carefully integrate data from disparate

sources to build a data warehouse in alignment with your existing business processes. Often, this data follows a Third Normal Form (3NF) or Data Vault model. Specifying primary and foreign key constraints allows end users to understand table relationships when using Unity Catalog. Your data warehouse should serve as the single source of truth for your data marts.

The data warehouse itself is schema-on-write and atomic. It is optimized for change, so you can quickly modify the data warehouse to match your current needs when your business processes change or evolve.

Gold layer

The gold layer is the presentation layer, which can contain one or more data marts. Frequently, data marts are dimensional models in the form of a set of related tables that capture a specific business perspective.

The gold layer also houses departmental and data science sandboxes to enable self- service analytics and data science across the enterprise. Providing these sandboxes and their own separate compute clusters prevents the Business teams from creating copies of data outside the lakehouse.

Layer	Purpose	Data Characteristics	Use Cases / Workloads
Silver	Curated, structured, and integrated data	Cleaned, transformed, partially structured data.	Data science, feature engineering, data warehousing, BI, and reporting.
Gold	Presentation-ready, business-specific data	Fully transformed, optimized for reporting/queries.	Business intelligence, dashboards, decision-making, and self-service analytics.

Bronze as a source:

- **ETL Jobs:** Raw data from the bronze layer is transformed and cleaned in the silver layer.
- **Data Science:** In cases where data scientists need raw data to explore, analyze, and develop new features for machine learning.

Gold as a source:

- **BI and Executive Reporting:** Gold layer is the primary source for business leaders looking for clear, high-level metrics and operational insights.

- **Self-Service Analytics:** Users from various departments can access this layer for business decision-making and detailed, but easy-to-understand analytics.

Identify the improvement in data quality in the data lakehouse over the data lake.

The data lakehouse architecture improves data quality over the traditional data lake by incorporating features typically associated with data warehouses while retaining the

flexibility and scalability of a data lake. Here's how the data quality improves in a data lakehouse compared to a data lake:

1. Structured, Curated Data vs. Raw Data

- Data Lake: In a data lake, data is typically stored in its **raw, uncurated** form, meaning it can be semi-structured, unstructured, or even incomplete. There's minimal or no transformation applied to the data as it is ingested, leading to potential quality issues such as duplicates, missing values, and inconsistencies.
- Data Lakehouse: In a lakehouse, data undergoes cleaning, structuring, and transformation as it moves through different layers (bronze, silver, and gold). This ensures that by the time it is queried or used for reporting, it is of much higher
  quality and has been integrated and validated. The silver and gold layers ensure that the data is well-structured, consistent, and business-ready.

2. Data Integrity and Governance

- Data Lake: Data lakes typically don't enforce data integrity constraints or governance rules. There's often no clear tracking of how data has been modified, leading to potential quality issues, like inconsistent schema or dirty data.
- Data Lakehouse: A lakehouse incorporates features of data governance, including enforcing schema-on-write and the ability to track changes using technologies like Delta Lake. This provides data integrity by applying consistent rules on data ingestion, transformations, and updates. Additionally, the use of ACID transactions (atomicity, consistency, isolation, and durability) ensures that updates to the data are reliable and consistent, preventing issues like partial updates or data corruption.

3. Unified Data Storage and Querying

- Data Lake: In a data lake, data is often stored in raw formats like JSON, Avro, or Parquet. While this allows for flexibility, querying and analyzing raw data is
  difficult and prone to errors, especially when working with heterogeneous data sources. There is no built-in mechanism for managing different versions or
  tracking data lineage.
- Data Lakehouse: A lakehouse unifies data storage and querying, making it easier to perform high-quality analytics on large datasets. By using technologies like Delta Lake or similar, a lakehouse provides capabilities like version control, time travel, and data lineage, allowing users to see exactly how data has evolved over time. This ensures that data can be traced back to its source and any errors can be easily identified and fixed, greatly improving data quality.

4. Performance Optimizations

- Data Lake: While data lakes are designed to scale to massive datasets, they lack the optimizations necessary for fast, efficient querying. This can result in poor query performance, especially as the volume of data grows, and lead to errors in reporting and analysis due to slow or incomplete data processing.
- Data Lakehouse: A lakehouse introduces performance optimizations such as indexes, caching, and optimized file **formats** (like Parquet or ORC). These improvements allow for faster query performance and reliable analytics on large datasets. The ability to manage both batch and real-time processing also means that data can be refreshed and accessed more quickly, improving the overall quality of insights derived from the data.

5. Support for Consistent Data Models

- Data Lake: Data lakes typically store a variety of raw formats from multiple sources without any enforced schema or structure. This can lead to difficulties in ensuring data consistency, especially when integrating data from disparate sources.
- Data Lakehouse: The data lakehouse architecture can support structured models (e.g., dimensional models or normalized models) in the gold layer, improving the consistency and clarity of data. It enforces data models that align with business processes, making the data more predictable, reliable, and meaningful for reporting, analytics, and decision-making.
-

6. Data Quality Monitoring and Validation

- Data Lake: In a data lake, there's minimal support for data quality monitoring, which can make it difficult to ensure that the data being ingested and stored
  meets quality standards. There’s a lack of tools for validating data quality during ingestion.
- Data Lakehouse: A lakehouse provides data validation and monitoring mechanisms, including the ability to automatically check for errors,
  inconsistencies, and missing data. This allows for automated quality checks as data enters the system and helps detect issues earlier in the data pipeline, improving overall data quality.

7. ACID Transactions and Data Consistency

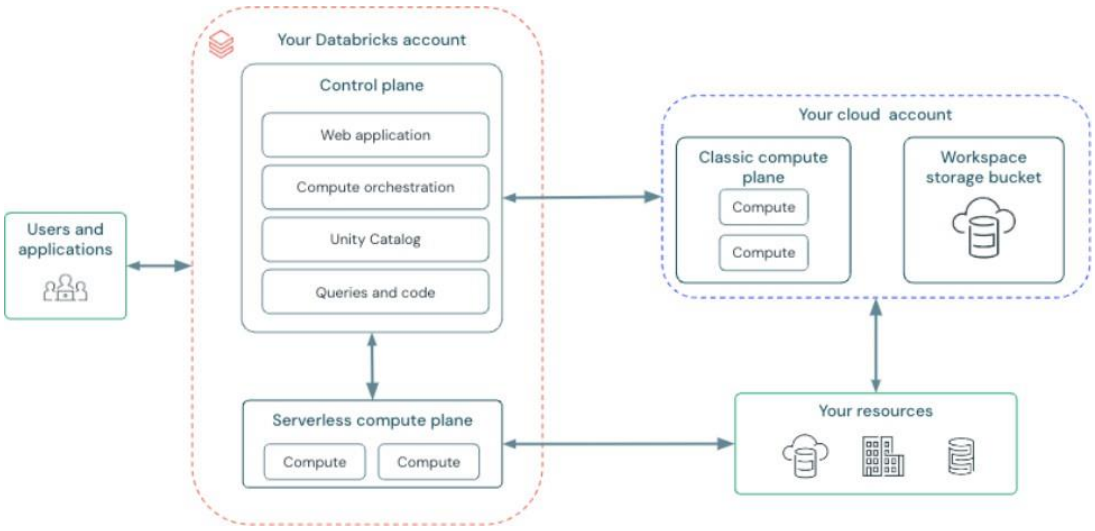
- Data Lake: Data lakes generally do not support ACID transactions, meaning that operations like data updates, deletions, and inserts might not be consistent or reliable. This lack of transactional guarantees can lead to data inconsistencies and errors.
- Data Lakehouse: A lakehouse uses ACID transaction support, often provided by Delta Lake or similar frameworks. This means that updates to the data are guaranteed to be atomic, consistent, isolated, and durable. As a result, any changes to the data are consistent across the system, ensuring data quality and preventing issues such as partial updates or corrupted data.

8. Advanced Data Transformations and Pipelines

- Data Lake: Transformations in a data lake are often complex and need to be managed manually or through external processing frameworks (like Apache Spark). This can lead to errors if transformations are not carefully executed, especially when dealing with raw or unclean data.
- Data Lakehouse: Data lakehouses facilitate more advanced data transformations through structured workflows, often providing better integration with ETL tools and data pipelines. The silver and gold layers allow for easy transformations that ensure data is cleansed and standardized before reaching users, making data much more reliable.

• **Identify elements of the Databricks Platform Architecture, such as what is in the data plane versus the control plane and what resides in the customer's**
cloud account.

The Databricks platform architecture is designed to provide a unified, cloud-native environment for data engineering, data science, and analytics. It is structured into two primary planes: the **Control Plane** and the **Data Plane**. Additionally, certain components reside in the customer's cloud account (e.g., AWS, Azure, or GCP). Below is a detailed explanation of these elements:



1.**Control Plane**

The Control Plane is the management layer of the Databricks platform, responsible for backend services and user interactions. It is fully managed by Databricks and operates within Databricks' cloud infrastructure. Key components include:

- **Web Application Interface**: Provides the user interface for managing Databricks resources, such as notebooks, jobs, and clusters.
- **Metadata Storage**: Manages workspace configurations, job schedules, and cluster metadata.
- **Cluster Management Services**: Handles the creation, scaling, and termination of compute clusters.
- **REST APIs**: Enables programmatic interaction with Databricks

resources, such as creating or managing jobs and clusters.

The Control Plane is shared across multiple customers within a region but ensures isolation through security measures like encryption and network controls.

2. Data Plane

The Data Plane is where data processing occurs. It is divided into two types based on the compute resources used: **Classic Compute Plane** and **Serverless Compute Plane**.

a. Classic Compute Plane

- **Location**: Resides in the customer's cloud account (e.g., AWS, Azure, or GCP).
- **Isolation**: Each customer's environment is naturally isolated because it runs within their own virtual network (VNet) and subscription.
- **Use Cases**: Ideal for long-running, production workloads requiring strict network and security controls.
- **Networking**: Requires configuration of private endpoints, firewalls, and VNets for secure communication with the Control Plane.

b. Serverless Compute Plane

- **Location**: Operates within Databricks' managed infrastructure but is created in the same region as the customer's workspace.
- **Isolation**: Provides multi-layered security, including network boundaries and cluster-level isolation, to ensure customer data remains secure.
- **Use Cases**: Suitable for ad-hoc queries, lightweight workloads, and scenarios requiring minimal operational overhead.

4. Security and Isolation

- **Control Plane to Data Plane Communication**:
  - Traffic between the Control Plane and Data Plane typically traverses the public internet. However, private connectivity options like AWS PrivateLink or Azure Private Link can be used to route traffic through the cloud provider's backbone network, enhancing security.
- **Multi-Tenant Isolation**:

- The Control Plane is shared across customers but employs encryption, network controls, and private endpoints to ensure isolation.

- **Data Protection**:

- Data in transit is encrypted using TLS 1.2 or higher, while data at rest is protected using AES-256 encryption.

5. Summary

The Databricks architecture is designed to balance flexibility, security, and performance. The Control Plane handles management and user interactions, while the Data Plane processes data in either a customer-managed (Classic Compute) or Databricks-managed (Serverless Compute) environment. Key components, such as workspace storage and classic compute resources, reside in the customer's cloud account, ensuring control and compliance.

● Differentiate between all-purpose clusters and jobs clusters.

Databricks provides two types of clusters to cater to different use cases: **All-Purpose Clusters** and **Jobs Clusters**. Each type is optimized for specific workloads and user requirements. Below is a detailed differentiation between the two:

1. All-Purpose Clusters

All-Purpose Clusters are designed for interactive and ad-hoc workloads, making them ideal for data exploration, development, and collaboration.

- **Primary Use Case**:

- Interactive data analysis, development, and debugging.

- Used by data scientists, data engineers, and analysts for ad-hoc queries, notebook development, and exploratory data analysis.

- **Cluster Management**:

- Manually created and managed by users through the Databricks UI, CLI, or REST API.

- Users can start, stop, and terminate clusters as needed.

- **Lifetime**:
  - Long-running clusters that persist until explicitly terminated by the user.
  - Ideal for scenarios where users need continuous access to compute resources.

- **Cost Implications**:
  - Since these clusters are long-running, they may incur higher costs if left idle.
  - Users are responsible for monitoring and terminating unused clusters to optimize costs.

- **Access and Sharing**:
  - Multiple users can attach to the same All-Purpose Cluster, making it suitable for collaborative work.
  - Notebooks and jobs can share the same cluster, enabling real-time collaboration.

2. Jobs Clusters

Jobs Clusters are optimized for automated, scheduled, or non-interactive workloads, such as production ETL pipelines, scheduled reports, or batch processing.

- **Primary Use Case**:
  - Running automated, non-interactive workloads.
  - Ideal for production jobs, scheduled tasks, and batch processing.

- **Cluster Management**:
  - Automatically created and managed by Databricks when a job is triggered.
  - Clusters are terminated automatically after the job completes, ensuring cost efficiency.

- **Lifetime**:
  - Short-lived clusters that exist only for the duration of the job.

- Clusters are spun up when the job starts and terminated once the job finishes (successfully or unsuccessfully).

- **Cost Implications**:
  - More cost-effective for automated workloads since clusters are only active during job execution.
  - No risk of idle clusters incurring unnecessary costs.

- **Access and Sharing**:
  - Not intended for interactive use or collaboration.
  - Each job runs in its own isolated cluster, ensuring reproducibility and isolation.

- **Example Use Cases**:
  - Running scheduled ETL pipelines to process and transform data.
  - Executing machine learning model training or batch inference jobs.
  - Generating and delivering scheduled reports or dashboards.

3. Key Differences Summary

Feature	All-Purpose Clusters	Jobs Clusters
Primary Use Case	Interactive development and analysis	Automated, non-interactive workloads
Cluster Management	Manually created and managed by users	Automatically created and managed by Databricks
Lifetime	Long-running; persists until manually terminated	Long-running; persists until manually terminated
Cost Efficiency	May incur higher costs if left idle	Cost-effective; no idle cluster costs
Access and Sharing	Multiple users can attach and collaborate	Isolated; not intended for interactive use
Example Use Cases	Notebook development, ad-hoc queries	Scheduled ETL, batch processing, ML jobs

This differentiation provides a clear understanding of when and how to use All-Purpose Clusters

versus Jobs Clusters in Databricks. It can be directly incorporated into your documentation to guide users in selecting the appropriate cluster type for their workloads.

- **Identify how cluster software is versioned using the Databricks Runtime.**

In Databricks, cluster software is versioned through the **Databricks Runtime**, which is a curated environment tailored for data processing and analytics. Each Databricks Runtime version is identified by a specific numbering system, such as 14.3 LTS or 15.0, indicating the sequence and type of release. These versions are released regularly and include updates that improve usability, performance, and security.

Databricks Runtime versions are released on a regular basis, and some versions are designated as Long Term Support (LTS) releases, indicated by an "LTS" qualifier (e.g., 14.3 LTS). LTS versions are supported for an extended period, providing a stable environment for critical workloads.

Each Databricks Runtime version includes updates that improve the usability, performance, and security of big data analytics. These updates may include new features, performance enhancements, and security patches

To determine the Databricks Runtime version of a cluster, you can access the cluster's configuration settings within the Databricks workspace. Additionally, during cluster initialization, the environment variable DATABRICKS_RUNTIME_VERSION can be used to programmatically retrieve the runtime version, although its availability may vary depending on the cluster's setup.

1. How Clusters Can Be Filtered to View Those Accessible by the User

In Databricks Lakehouse Platform, clusters can be filtered to show only those that are accessible by the user based on permissions and roles. Here's how you can filter clusters:

Steps to Filter Clusters:

1. **Navigate to the Clusters Page**:
    - Go to the **Clusters** section in the Databricks workspace.
2. **Use the Filter Option**:
    - At the top of the clusters list, you'll find a **filter bar**.
    - You can filter clusters by:
        - **Cluster Name**: Enter the name of the cluster.
        - **Cluster State**: Filter by running, terminated, or pending clusters.
        - **Created By**: Filter clusters created by a specific user.
        - **Access Level**: Filter clusters based on your access level (e.g., "Can Attach To" or "Can Manage").
3. **View Accessible Clusters**:
    - By default, the clusters list shows clusters that the user has access to based on their permissions.
    - If you have **Admin** or **Workspace Admin** privileges, you can see all clusters in the workspace.
    - If you are a regular user, you will only see clusters that you have permission to access (e.g., clusters you created or clusters shared with you).
4. **Permissions**:
    - **Can Attach To**: Users with this permission can attach notebooks to the cluster.
    - **Can Manage**: Users with this permission can start, stop, or delete the cluster.
    - **Can Restart**: Users with this permission can restart the cluster.

Example:

- If you are a data scientist, you might only see clusters that you created or clusters shared with your team.
- If you are an admin, you can see all clusters in the workspace and filter them by

owner, state, or name.

2. How Clusters Are Terminated and the Impact of Terminating a Cluster Terminating a Cluster:

1. **Manual Termination**:
    - Go to the **Clusters** page.
    - Find the cluster you want to terminate.
    - Click the **three dots (⋮)** next to the cluster name and select **Terminate**.
    - Confirm the termination.
2. **Automatic Termination**:
    - Clusters can be configured to terminate automatically after a period of inactivity (e.g., 30 minutes, 1 hour).
    - This is set in the cluster configuration under **Auto Termination**.

Impact of Terminating a Cluster:

1. **Data and Jobs**:
    - Terminating a cluster **does not delete data** stored in DBFS (Databricks File System) or external storage (e.g., S3, ADLS).
    - Any jobs or notebooks attached to the cluster will stop running.
2. **Cluster State**:
    - The cluster will transition to the **Terminated** state.
    - You can restart the cluster later, but it will start from scratch (no cached data or state).
3. **Cost Implications**:
    - Terminating a cluster stops billing for compute resources.
    - If the cluster is part of a job, the job will fail unless a new cluster is provisioned.
4. **Recreation**:
    - If you need the cluster again, you must recreate it or restart it (if it's configured to allow restarts).

3. Scenario in Which Restarting the Cluster

Will Be Useful Scenario:

- **Cluster Performance Degradation**:
    - Over time, a cluster may experience performance degradation due to:
        - Memory leaks in applications.
        - Accumulation of temporary files or cached data.
        - Resource contention (e.g., too many users attaching to the cluster).

Why Restarting Helps:

1. **Clears Memory and Cache**:
    - Restarting the cluster clears all in-memory data and caches, which can resolve memory-related issues.
2. **Resets State**:
    - Restarting resets the cluster to a clean state, eliminating any temporary issues caused by long-running processes.
3. **Improves Performance**:
    - After a restart, the cluster will have access to fresh resources, improving performance for subsequent tasks.

Steps to Restart a Cluster:

1. Go to the **Clusters** page.
2. Find the cluster you want to restart.
3. Click the **three dots (⋮)** next to the cluster name and select **Restart**.
4. Confirm the restart.

Example Use Case:

- A data engineering team is running a daily ETL job on a cluster. Over time, they notice that the job is taking longer to complete due to memory issues. Restarting the cluster before the next job run ensures that the cluster starts fresh, improving performance and reliability.

1. **Using Multiple Languages Within the Same Notebook**

   Databricks notebooks support multiple programming languages (e.g., Python, SQL, Scala, R) within the same notebook. To use multiple languages:

    - Each cell in the notebook can be set to a specific language.
    - Use the %<language> magic command at the beginning of a

cell to specify the language. For example:

- %python for Python
- %sql for SQL
- %scala for Scala
- %r for R

- The notebook will automatically interpret the code in the specified language for that cell.

---

2. Running One Notebook from Within Another Notebook

You can run one notebook from another notebook using the %run magic command:

- Use %run /path/to/notebook to execute the contents of another notebook.
- The executed notebook's variables, functions, and outputs will be available in the
calling notebook.
- This is useful for modularizing code and reusing common logic across notebooks.

---

3. Sharing Notebooks with Others

Databricks provides several ways to share notebooks:

- **Workspace Sharing**: Share notebooks directly within your Databricks workspace by granting permissions to specific users or groups.
- **Export/Import**: Export notebooks as .dbc or .ipynb files and share them externally. Recipients can import these files into their Databricks environment.
- **Git Integration**: Sync notebooks with Git repositories (e.g., GitHub, GitLab) to collaborate and share code with others.
- **Dashboard and Reports**: Share insights by publishing notebook outputs as dashboards or reports to stakeholders.

Databricks Repos and CI/CD Workflows

1. How Databricks Repos Enables CI/CD Workflows

Databricks Repos integrates Git repositories with Databricks workspaces, enabling seamless version control and CI/CD (Continuous Integration/Continuous Deployment) workflows. Here's how it facilitates CI/CD:

- **Git Integration**: Databricks Repos allows users to connect their workspace to Git repositories (e.g., GitHub, GitLab, Bitbucket). This enables version control for notebooks, scripts, and other code artifacts.
- **Branching and Pull Requests**: Users can create branches, make changes, and submit pull requests directly from the Databricks workspace. This aligns with standard Git workflows and facilitates collaboration.
- **Automated Testing and Deployment**: By integrating with CI/CD tools (e.g., Jenkins, GitHub Actions, Azure DevOps), changes in the Git repository can trigger automated testing and deployment pipelines. For example:
    - A pull request can trigger unit tests for notebooks.
    - Merging to the main branch can deploy the updated code to production environments.
- **Environment Consistency**: Repos ensures that the same version of code is used across development, staging, and production environments, reducing inconsistencies and errors.

---

2. Git Operations Available via Databricks Repos

Databricks Repos supports the following Git operations:

- **Clone a Repository**: Connect a Git repository to the Databricks workspace.
- **Create and Switch Branches**: Create new branches or switch between existing branches.
- **Commit Changes**: Commit changes made to notebooks or files directly from the workspace.
- **Push and Pull**: Push local changes to the remote repository or pull the latest changes from the remote repository.
- **Merge Branches**: Merge changes from one branch to another.

- **Create Pull Requests**: Create pull requests to merge changes into the main branch.
- **Revert Changes**: Revert to a previous commit or branch state.

3. Limitations in Databricks Notebooks Version Control (Without Repos)

Before Databricks Repos, version control for notebooks was limited. Here are the key limitations compared to Repos:

- **No Native Git Integration**: Without Repos, notebooks could not be directly linked to Git repositories. Users had to manually export and import notebooks to version control systems.
- **Limited Collaboration**: Changes made to notebooks were not tracked in a centralized Git repository, making collaboration and code reviews difficult.
- **No Branching or Pull Requests**: Users could not create branches or pull requests directly from the Databricks workspace.
- **Manual Deployment**: Deploying changes to production required manual steps, increasing the risk of errors.
- **Lack of Traceability**: Without Git integration, it was harder to track changes, revert to previous versions, or audit code history.

Extract data from a single file and from a directory of files:

When working with datasets of any size in DataBricks, one of the first and most important steps in an ETL pipeline is the extraction of data from various sources. Apache Spark, integrated with DataBricks, makes it easier to load data from a variety of file formats like CSV, JSON, Parquet and Delta.

Extracting Data from a Single File:

In the case where required data is present in a single file (be it CSV, JSON, Parquet, or Delta), Apache Spark's `spark.read` method is the most appropriate. It supports reading data from multiple file formats, thereby helping to load data efficiently into a DataFrame for performing further processing.

Assuming data is in a CSV file named ` mock_data.csv `, the following

code snippet is used to load the data into a dataframe `df = spark.read.options(“header”, “true”).csv(“path/to/mock_data.csv”)`. Here, the `.options(“header”, “true”)` parameter tells Spark that the first row of the data contains the headers for column name. Another important parameter of `.options()` method is `inferSchema`. If enabled, Spark tries to automatically find the datatype of each column automatically, whereas if the schema is already know it can be disabled. If the dataset is of type JSON, use `spark.read.json(“path/to/file”)`, and for a parquet file use `spark.read.parquet(“path/to/file”)`. Another method for reading is `spark.read.format(“parquet”).load(“path/to/file”)`. Both methods are correct, and do not offer any advantage over each other.

Extracting Data from a Directory of Files:

In many scenarios, especially when the data is very large, instead of storing the data into a single file it will be spread across multiple files. This is very useful for partitioned data, which makes accessing it later much easier and more efficient. To read multiple files, path to the directory containing the files is given as input to the `spark.read` method.

For example, to read multiple CSV files stored in a specific folder and load them into a dataframe, use the following code snippet `df = spark.read.options(“header”, “true”).csv(“path/to/data/folder/with/CSV/files”)`. Or, `spark.read.format(“fileformat”).load(“path/to/data/folder/with/CSV/files”)` can also be used to load data stored across multiple files. Wildcard characters can be used when specifying the path inorder to match a specific pattern in folder name or file name.

While handling large datasets in a platform like DataBricks, one of the key benefits is the ability to scale and handle large datasets efficiently. While extracting data from a directory of files, Apache Spark will distribute the file reading process across multiple clusters, thereby reading multiple files in parallel. This distributed approach ensures that Spark can handle a large volume of data quickly and efficiently.

In addition to this, the reading process can be further optimized by using partitioning (dataset is partitioned by column to speed up querying and processing) and clustering (frequently accessed data is cached in memory to avoid repetitive reads, thereby improving performance).

Identify the prefix included after the FROM keyword as the data type:

In Apache Spark, the FROM keyword plays an essential role in extracting data during SQL queries. It specifies the source from which data is to be fetched. Apache Spark supports various data types like CSV, JSON, Parquet, Delta and even tables and views stored within a Spark Session. Identifying the correct data type after the FROM keyword is important as format of the data impacts the way Spark reads, processes, and queries the data.

Various Methods of Reading Data:

Apache Spark allows reading data from various sources like files, tables, views and external sources.

1. File-Based data sources:

   Each different file format has a specific prefix hat can be used in SQL queries or with Dataframe API to tell Spark how to handle the data.

   When working with CSV files, the data type can be explicitly mentioned using the `CSV` keyword. Example `SELECT * FROM csv("path/to/csv/file")`.

   When working with JSON files, the query will be like `SELECT * FROM json("path/to/json/file")`.

   And when working with Parquet file, the query will be like `SELECT * FROM parquet("path/to/parquet/file")`.

2. Delta Tables:

   Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark and big data workloads.

   While working with Delta tables, the prefix "delta" is used to indicate source format. Its query will be like `SELECT * FROM delta("path/to/delta/file")`.

3. Views:

   When working with temporary views, they are simply referred my their names. Example:
   `SELECT * FROM temp_view`.

   For creating a temporary view, the `createOrReplaceTempView()` method is used. Example:
   `df. createOrReplaceTempView("temp_view")`, where `df` is the dataframe containing required data, while ` temp_view ` is the name of the temporary view created through the above code.

4. External Sources:

Other than local files and views and delta tables, Apache Spark is capable of reading data from other sources like Apache HDFS, AWS S3, Azure Blob Storage, or Google Cloud Storage. To access files in these sources, simply their paths are given after the `FROM` keyword. Prefix after the `FROM` keyword is dependent upon the storage system.

Example, reading a file from from DataBricks File System: `SELECT * FROM dbfs:/path/to/file`

Identifying the prefix after the FROM keyword is crucial for telling Spark how to interpret the data source correctly. It defines the data type, whether it's a file, Delta table, or a view. Knowing the correct prefix allows Spark to correctly parse and load the data, making it ready for further
transformation and analysis steps.

Create a view, a temporary view, and a CTE as a reference to a file:

An important feature of Apache Spark is the ability to create views, temporary views, and Common Table Expressions, which allow users to structure and reuse queries in an organized and efficient manner. They can be used to reference data stored in various types of files and external sources without having to repeatedly load the data for each individual query. Views and CTEs provide an abstraction layer, hiding the complexities of loading and storing, making it easier to manage and analyze the data.

Creating a View:

A view is a saved SQL query that can be referred to as a table in subsequent queries. It is a virtual table based on the result of a query. They are persistent and stored in the DataBricks metastore, unless explicitly dropped. Views can be accessed across multiple sessions and by various users with the right permissions.

Example 1:

```
CREATE VIEW view_data AS
SELECT * FROM
csv("path/to/file")

SELECT * FROM view_data
WHERE col1="value1"
```

Creating a Temporary View:

A temporary view is similar to a regular view but is session-scoped, meaning the view will only exist within the lifecycle of a Spark session, and will be dropped when the session ends. Temporary Views are useful for
intermediate data processing or in situations where it is not necessary to store the data permanently. The
`createOrReplaceTempView()` method is used for creating temporary views.

Example 1:

```
df = spark.read.csv("path/to/csv/file")
df.createOrReplaceTempView("temp_view_name")
```

```
SELECT *
FROM temp_view_name
```

Creating a Common Table Expression:

A CTE is a temporary result set that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. Unlike views, CTEs only exist within the context of the query containing it. They are particularly useful when a complex query needs to be broken down into simpler components or reuse the same subquery multiple times within the same query.

Example 1:

```
WITH dataCTE AS (
    SELECT *
    FROM csv("path/to/csv/file")
)
SELECT * FROM dataCTE;
```

CTEs are powerful while dealing with complex transformations and aggregations. They help with better organization and readability of queries, especially when the same result set needs to be used in different parts of the same query.

<u>Identify that tables from external sources are not Delta Lake tables:</u>

In ETL pipeline using Apache Spark, it is important to understand the distinction between Delta Lake tables and tables form external sources. Delta Lake tables provide significant advanced features like ACID transactions, schema enforcement, time travel and the ability to handle large-scale batch and streaming data workloads – all of which are not available in non-Delta tables.

Thus, it is important to identify such tables (non-Delta tables) inoder to understand their limitations and determine necessary actions to be taken to optimize their use.

There are multiple ways to check if a table is a Delta Lake table or not:

a. Using Spark SQL

   The following query is used to query the table's metadata `DESCRIBE DETAIL "path/to/table"`. If the table is a Delta table, then the `provider` field in the output will be "delta".

b. Using File System Inspection

   Delta Lake tables store metadata in a `_delta_log` directory. This directory will be present in the table's storage location [table's storage location can be found out using `dbutils.fs.ls("path/to/table")`]. If the table is a Delta Table, there will be a `_delta_log` directory containing transaction logs.

c. Using Delta Lake API

   Delta Lake API can also be used to check if a table is a Delta table or not.

```
from delta.tables import DeltaTable

try:
    delta_table = DeltaTable.forPath(spark, "path/to/table")
    print("This is a Delta Lake table")
except Exception as e:
print("This is not a Delta Lake table")
```

Handling non-Delta tables:

Even of a table is a non-Delta table, it can be converted into Delta format inorder to take advantage of
Delta Lake's features. This can be done using the `CONVERT TO DELTA` command: `CONVERT TO DELTA parquet."path/to/parquet/file"`.

Instead of converting the table permanently, it is also possible to read the non-Delta and then write it as a Delta table:

```
non_delta_df = spark.read.parquet("path/to/parquet/file")
non_delta_df.write.format("delta").save("path/to/delta/table/save/location")
```

Benefits of using Delta Lake:

- ACID Transactions:
  - Delta Lake provides transactional guarantees. This also ensures data integrity during concurrent read operations. For non-Delta files, additional mechanisms or tools might be needed for this.
- Performance Optimizations:
  - Delta Lake tables support optimization techniques like Z-order indexing and data skipping, which improves query performance.
- Data Integrity:
  - Delta Lake tables enforce schema checks and can automatically evolve the schema. Whereas, non-Delta tables might require manual schema management, and inconsistencies in data can lead to issues while running the query.
- Time Travelling and Auditing:
  - By using Delta Lake's time travel feature historical versions of the data can be queried, which is not possible in non-Delta tables unless some sort of versioning tools are used.

Create a table from a JDBC connection and from an external CSV file.

Create a Table from a JDBC Connection

Use Spark's JDBC data source to connect to databases like PostgreSQL, MySQL, etc.

Example: Read from PostgreSQL

```
# Configure JDBC connection properties
jdbc_url = "jdbc:postgresql://<host>:<port>/<database>"
connection_properties = {
    "user": "<username>",
    "password": "<password>",
    "driver": "org.postgresql.Driver"
}

# Read data from a JDBC table
jdbc_df = spark.read.jdbc(
    url=jdbc_url,
    table="<table_name>",
    properties=connection_properties
)

# Write to a Delta Lake table (managed or external)
jdbc_df.write.format("delta").saveAsTable("jdbc_table")  # Managed table
# jdbc_df.write.format("delta").save("/path/delta/jdbc_table")  # External table
```

Notes:

- Driver JAR: Attach the JDBC driver (e.g., postgresql-42.7.3.jar) to your cluster.
- Partitioning: For large tables, use partitionColumn, lowerBound, upperBound, and numPartitions to parallelize reads.

Create a Table from an External CSV File

Read CSV files stored in cloud storage (e.g., S3, ADLS, GCS) or DBFS.

Example: Read CSV from DBFS

```python
# Read CSV with header and inferred schema
csv_path = "dbfs:/path/file.csv"
csv_df = spark.read \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .csv(csv_path)

# Explicit schema definition (recommended for production)
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True)
])


csv_df = spark.read \
    .schema(schema) \
    .option("header", "true") \
    .csv(csv_path)

# Write to a Delta Lake table

# Managed table
csv_df.write.format("delta").saveAsTable("csv_table")

# External table
csv_df.write.format("delta").save("/path/delta/csv_table")
```

To query the table using Spark SQL

```python
spark.sql("SELECT * FROM jdbc_table").show()
spark.sql("SELECT * FROM csv_table").show()
```

Deduplicate rows from an existing Delta Lake table

There are different methods for deduplication and they are:

- Deduplication using dropDuplicates()

```
df = spark.read.format("delta").load("/path/to/delta/table")
```

- Window functions approach for keeping specific records.
- Merge operation for more efficient deduplication on large tables.

Deduplication using dropDuplicates()

This is the simplest approach,to use Spark's dropDuplicates() function combined with overwrite mode

```
# Read the existing Delta table



# Drop duplicates (specify the columns that define uniqueness)
df_deduplicated = df.dropDuplicates(["id", "timestamp"])  # adjust columns as needed
                                -
# Overwrite the existing table with deduplicated data
df_deduplicated.write.format("delta").mode("overwrite").save("/path/to/delta/table")
```

Note: For large tables this may be inefficient since its rewrites the entire table.

Window functions approach for keeping specific records

This is used when we need more control over which duplicate to keep.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number, col

# Read the Delta table
df = spark.read.format("delta").load("/path/to/delta/table")

# Define window specification   partitioned by the columns that define uniqueness
#
Window_spec = Window.partitionBy("id").orderBy(col("timestamp").desc())

# Add row numbers within each partition
df_with_row_numbers = df.withColumn("row_num", row_number().over(window_spec))

# Keep only the first record of each group (row_num = 1)
df_deduplicated = df_with_row_numbers.filter(col("row_num") == 1).drop("row_num")

# Write back to the Delta table
df_deduplicated.write.format("delta").mode("overwrite").save("/path/to/delta/table")
```

Using Merge Operation for Efficient Deduplication

```
from delta.tables import DeltaTable

# Create a view of the Delta table


delta_table  = DeltaTable.forPath(spark, "/path/to/delta/table")

# Create a deduplicated view of the source data
source_df = spark.read.format("delta").load("/path/to/delta/table")
deduplicated_df = source_df.groupBy("id").agg(
    *[first(col).alias(col) for col in source_df.columns]
)

  Perform a merge operation
delta_table.alias("target").merge(
    deduplicated_df.alias("source"),
    "target.id = source.id"
).whenMatchedUpdateAll().whenNotMatchedInsertAll().execute()
```

> Note: best approach for large tables.

Best Practices

Time Travel: Before performing deduplication, consider creating a checkpoint:

```
current_version = spark.read.format("delta").load("/path/to/delta/table").toPandas()
print(f"Current version: {current_version}")
```

Vacuum Protection: Ensure your retention period is adequate before vacuuming:

```
spark.conf.set("spark.databricks.delta.retentionDurationCheck.enabled", "false")
spark.sql(f"VACUUM delta.`/path/to/delta/table` RETAIN 7 DAYS")
```

Optimize After Deduplication: Consider optimizing the table after deduplication:

```
spark.sql(f"OPTIMIZE delta.`/path/to/delta/table`")
```

values varies depending on its

Handling NULL Values with COUNT in Databricks SQL

Overview

In Databricks SQL, the behavior of the `COUNT` function in relation to `NULL` usage:

- **`COUNT(*)`** : Counts all rows in the group, including those containing `NULL` values.
- **`COUNT(expr)`** : Counts all rows for which the expression `expr` is not `NULL` , effectively skipping `NULL` values.

Syntax

```
-- Count all rows, including those with NULLs
COUNT(*)

-- Count rows where the expression is not NULL
COUNT(expr)
```

Examples

Consider a table `person` with the following data:

name	age
Alice	25

Bob	NULL
Charlie	30
David	35
Eve	NULL

1. Counting all rows

```sql
SELECT COUNT(*) AS total_rows FROM person;
```

Result:

total_rows
5

This counts all rows, including those where `age` is `NULL`

2. Counting non- `NULL` values in a specific column

```
SELECT COUNT(age) AS non_null_ages FROM person;
```

Result:

non_null_ages
3

This counts only the rows where

3. Counting distinct non-

`age` is not `NULL`

`NULL` values

distinct_non_null_ages
3

```
SELECT COUNT(DISTINCT age) AS distinct_non_null_ages FROM person;
```

Result:

-

This counts the distinct non- `NULL` values in the `age` column.

Note

The `COUNT` function can also be used with the `FILTER` clause to count rows that meet specific conditions.

DATABRICKS DOCUMENTATION – ELT USING APACHE SPARK

1. Create a new table from an existing table while removing duplicate rows.
   - To create a new table from an existing one, removing duplicate rows in Spark (Databricks), use the dropDuplicates() function.

Code	existing_df = spark.table("existing_table") # Removing duplicate rows
	new_df = existing_df.dropDuplicates()
	
	# Create a new table in Databricks
	new_df.write.saveAsTable("new_table")

   - dropDuplicates():removes rows that are exactly the same across all columns.
   - saveAsTable():creates a new table from the DataFrame.

2. Deduplicate a row based on specific columns.
   - To deduplicate rows based on specific columns, you can use the dropDuplicates() function with a subset of columns.

Code	existing_df = spark.table("existing_table")
	
	# Deduplicate based on specific columns
	deduplicated_df = existing_df.dropDuplicates(["column1", "column2"])
	
	# Create a new table with the deduplicated rows
	deduplicated_df.write.saveAsTable("deduplicated_table")

   - dropDuplicates(["column1", "column2"]) removes rows that have duplicate values in column1 and column2, while retaining the first occurrence of each unique combination.

3. Validate that the primary key is unique across all rows.
   - You can use the row_number() function to assign a unique number to each row within a group (partitioned on the primary key). If any primary key has more than one row with row_number() values, it indicates duplicates.

Code

```
from pyspark.sql.window import Window
from pyspark.sql import functions as F

existing_df = spark.table("existing_table")

# Define a window specification that partitions by the primary key column
window_spec = Window.partitionBy("primary_key_column").orderBy("primary_key_column")

# Add a row number to each row based on the primary key column
df_with_row_num = existing_df.withColumn("row_num", F.row_number().over(window_spec))

# Filter rows where row_num > 1 (duplicates)
duplicates_df = df_with_row_num.filter(df_with_row_num["row_num"] > 1)

# Show any duplicates
duplicates_df.show()
```

- **Window specification (**Window.partitionBy("primary_key_column")**)**: This ensures that the row_number()is calculated within groups defined by the primary key.

- row_number().over(window_spec): This assigns a unique number to each row within the group.

- filter(df_with_row_num["row_num"] > 1): This filters out rows where the ROW_NUMBER()is greater than 1, indicating that the primary key has duplicates.

Section 2: ELT with Apache Spark

- Validate that a field is associated with just one unique value in another field.
- Validate that a value is not present in a specific field.
- Cast a column to a timestamp.

In Apache Spark, performing Extract, Load, and Transform (ELT) operations involves various data validation and transformation tasks.

1. Validate that a field is associated with just one unique value in another field

This task ensures that a specific field has a one-to-one relationship with another field.

- Ensure a field maps to only one unique value in another field.
- Useful for enforcing one-to-one relationships or preventing duplicate mappings.

To ensure that each value in one column (e.g., field1) corresponds to a unique value in another column (e.g., field2), you can perform the following steps:

- **Group by the first field and count occurrences of the second field:**

Eg:

 Group by 'field1' and count distinct 'field2' values

```
validation_df = df.groupBy('field1').agg(F.countDistinct('field2').alias('distinct_field2_count'))
```

 Filter to find 'field1' values associated with more than one 'field2' value

```
invalid_entries = validation_df.filter('distinct_field2_count > 1')
```

2. Validate that a value is not present in a specific field:

- Check that a specific value is not present in a given field.
- Helps maintain data integrity by excluding invalid or unwanted values.

To check that a specific value (e.g., 'invalid_value') is not present in a column (e.g., field1), you can use:

- **Filter the DataFrame to find rows where field1 equals 'invalid_value':**

Eg:

```
invalid_rows = df.filter(df['field1'] == 'invalid_value')
```

3. Cast a column to a timestamp:

- Convert a column to a timestamp data type for time-based operations.
- Essential for time-series analysis, event logging, or scheduling tasks.

To convert a column (e.g., date_string) from a string to a timestamp, you can use the to_timestamp function:

- **Using PySpark's to_timestamp function:**

Eg:

- Convert 'date_string' to timestamp

```
df = df.withColumn('timestamp_column', F.to_timestamp('date_string', 'yyyy-MM-dd HH:mm:ss'))
```

Section 2: ELT with Apache Spark

- Extract calendar data from a timestamp.
- Extract a specific pattern from an existing string column.
- Utilize the dot syntax to extract nested data fields.

1. Extract Calendar Data from a Timestamp

In Apache Spark, the timestamp type is used to represent points in time with a specific format. You can extract different calendar components (such as year, month, day, hour, minute, etc.) from a timestamp using built-in functions.

Key Functions:

- year(): Extracts the year from a timestamp.
- month(): Extracts the month from a timestamp.
- dayofmonth(): Extracts the day of the month from a timestamp.
- hour(): Extracts the hour from a timestamp.
- minute(): Extracts the minute from a timestamp.
- second(): Extracts the second from a timestamp.

```
from pyspark.sql.functions import year, month, dayofmonth, hour, minute, second

# Assuming 'df' is a DataFrame with a 'timestamp' column
df = df.withColumn("year", year("timestamp"))
df = df.withColumn("month",
```

month("timestamp")) df =

df.withColumn("day",

dayofmonth("timestamp")) df =

df.withColumn("hour",

hour("timestamp"))

df = df.withColumn("minute",

minute("timestamp")) df =

df.withColumn("second",

second("timestamp"))

2. Extract a Specific Pattern from an Existing String Column

When working with strings in Apache Spark, you often need to extract specific patterns from a column (e.g., extracting an email domain, phone number, etc.). Apache Spark provides the regexp_extract() function to perform regular expression matching and extraction.

Key Function:

- regexp_extract(): This function allows you to apply a regular expression and extract a specific portion of a string.

Syntax:

regexp_extract(column: Column, pattern: str, idx: int) -> Column

- column: The column from which you want to extract data.
- pattern: The regular expression pattern.
- idx: The index of the capture group to extract.

from pyspark.sql.functions import regexp_extract

Assuming 'df' is a DataFrame with an 'email' column

df = df.withColumn("domain", regexp_extract("email", r"@([a-zA-Z0-9.-]+)", 1))

3. Utilize the Dot Syntax to Extract Nested Data Fields

Apache Spark supports working with nested structures such as JSON, arrays, and maps. The dot syntax is used to extract fields from nested data.

Key Function:

- You can reference nested fields directly using the dot notation to navigate through JSON structures or other nested types.

Example: Assume you have a JSON column with a nested structure and you want to access fields like user.name or user.address.city:

```
from pyspark.sql.functions import col


# Example with nested JSON structure
df = df.withColumn("user_name",
col("user.name")) df =
df.withColumn("city",
col("user.address.city"))
```

In this example, col("user.name") references the name field within the nested user structure, and col("user.address.city") references the city field within the nested address structure.

Important Notes:

- When working with **JSON** or **nested structures**, you can apply dot notation directly on the column.
- If the field is within an array, you might need to use array functions such as getItem() or explode() to work with the elements.

For more on extracting fields from nested data and JSON in Spark, refer to the Databricks guide.

Benefits of Using Array Functions

Array functions in Databricks offer several key benefits:

1. **Simplified Data Manipulation**: Array functions make it easier to work with complex data structures by allowing you to perform operations directly on arrays. This includes functions like array_contains, array_distinct, array_except, and more.
2. **Performance Optimization**: By using array functions, you can reduce the need for multiple joins and aggregations, which can improve query performance. For example, instead of joining multiple

tables to get a list of unique elements, you can
use array_distinct to achieve the same result more efficiently.

3. **Enhanced Flexibility**: Array functions provide a wide range of operations, such as filtering (filter), transforming (transform), and aggregating (aggregate) elements within arrays. This flexibility allows for more complex data processing tasks to be performed directly within SQL queries.

Parsing JSON Strings into Structs

To parse JSON strings into structs in Databricks, you can use the from_json function. This function converts a JSON string into a struct based on a specified schema. Here's a more detailed example:

SELECT from_json('{"name":"John", "age":30}', 'name STRING, age INT') AS parsed_json;

In this example, the JSON string {"name":"John", "age":30} is parsed into a struct with
fields name (of type STRING) and age (of type INT). The from_json function can also handle more complex JSON structures and schemas.

Result of a Join Query

The result of a join query in Databricks depends on the type of join used. Here are some common join types and their results:

1. **Inner Join**: Returns only the rows that have matching values

   in both tables. SELECT a.*, b.*

   FROM table_a a

   INNER JOIN

   table_b b ON a.id =

   b.id;

This query will return rows where the id column matches in both table_a and table_b.

2. **Left Join (Left Outer Join)**: Returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table.

   SELECT a.*, b.*

```
FROM table_a a
LEFT JOIN table_b
b ON a.id = b.id;
```

This query will return all rows from table_a, and matching rows from table_b. If there is no match, columns from table_b will have NULL values.

3. **Right Join (Right Outer Join)**: Returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for columns from the left table.

```
SELECT a.*, b.*
FROM table_a a
RIGHT JOIN
table_b b ON a.id =
b.id;
```

This query will return all rows from table_b, and matching rows from table_a. If there is no match, columns from table_a will have NULL values.

4. **Full Outer Join**: Returns all rows when there is a match in either left or right table. Rows without a match in one of the tables will have NULL values for columns from that table.

```
SELECT a.*, b.*
FROM table_a a
FULL OUTER JOIN
table_b b ON a.id = b.id;
```

This query will return all rows from both tables, with NULL values for non-matching rows.

When to Use explode() vs. flatten()

Scenario for explode():

Breaking Data into Rows

Use explode()when you want to convert an array into multiple rows.

Example: Expanding Product Categories

Imagine an e-commerce store where each product belongs to multiple categories stored as an array.

product_id	categories
101	["Electronics", "Gadgets"]
102	["Books", "Fiction"]

If you want **one row per category**, use explode():

SELECT product_id, explode(categories) AS category FROM products;

Result:

product_id | category

---

101 | Electronics
101 | Gadgets
102 | Books
102 | Fiction

Each category now appears in its own row!

Scenario for flatten(): Merging

Nested Lists into One

Use flatten()when you have arrays inside arrays and want to merge them into a single array.

Example: Combining User Interests

Say you have a table storing user preferences, but they are grouped into multiple lists:

userid	interests
201	[["Music", "Sports"], ["Movies", "Travel"]]

To get a **single flat list** of interests:

```
SELECT user_id, flatten(interests) AS all_interests FROM users;
```

Result:

```
user_id | all_interests
  ----------------------

201    | ["Music", "Sports", "Movies", "Travel"]
```

This have a simple, merged array!

Key Differences

Function	Use Case
explode ()	
flatten ()	Convert an array **into rows** (e.g., listing all categories of a product separately)
	Merge **nested arrays** into a **single array** (e.g., combining user interests into one list)

The PIVOT clause is a powerful SQL feature that converts data from a long format (tall and narrow) to a wide format (short and wide).

Understanding Long vs. Wide Format

- **Long Format (Normalized Table)**
  Each row represents a unique observation, and categories are stored in a column.
- **Wide Format (Denormalized Table)**
  Categories become separate columns, making the data easier to read for reporting.

Example: Converting Long to Wide Format

Long Format (Before Pivoting)

year	quarter	region	sales
2018	1	east	100
2018	2	east	20
2018	3	east	40
2018	4	east	40
2019	1	east	120

Pivot Query

```
SELECT year, region, q1, q2, q3,
q4 FROM sales
PIVOT (SUM(sales) AS
  sales FOR quarter
  IN (1 AS q1, 2 AS q2, 3 AS q3, 4 AS q4));
```

Wide Format (After Pivoting)

year	region	q1	q2	q3	q4
2018	east	100	20	40	40

2019 east 120 - - -

Key Takeaway

The **PIVOT** clause efficiently **reshapes the data** by converting categorical values into column names, making it easier to analyze trends and generate reports.

A SQL User-Defined Function (SQL UDF) is a custom function written in SQL that extends the capabilities of SQL

queries by encapsulating logic within a reusable function. Unlike external UDFs written in languages like Python, Java, or Scala, SQL UDFs are defined entirely in SQL and are optimized by the SQL engine.

Key Characteristics of SQL UDFs:

- Defined using **SQL syntax**.
- Can **return scalar values or tables**.
- **Transparent to the query optimizer**, making them more performant.
- Can be **temporary or permanent**.
- Support **access control** for security.

Basic Example of a SQL UDF

Scalar SQL UDF

A function that returns a constant

value: CREATE FUNCTION

blue()

RETURNS STRING

COMMENT 'Returns the hex code for blue'

RETURN '0000FF';

Usage:

```
SELECT blue();  -- Returns '0000FF'
```

Table-Valued SQL UDF

A function that returns a table:

```
CREATE FUNCTION get_colors()

RETURNS TABLE(name STRING, rgb STRING)

RETURN SELECT name, rgb FROM colors;
```

Usage:

```
SELECT * FROM get_colors();
```

SQL UDFs simplify query logic, improve readability, and enhance performance compared to external UDFs.

1

1. Identifying the location of a function

In Databricks, functions can exist in various locations based on their type. These functions may be built-in, user-defined (UDFs), or stored in a specific catalog and schema when using Unity Catalog.

1.1 Types of Functions in Databricks

Functions in Databricks fall into three categories:

- Built-in Functions: Provided by Spark and Databricks like SUM(), LENGTH(), CURRENT_TIMESTAMP()
- User-Defined Functions (UDFs): Custom functions created by users in SQL, Python, or Scala.
- System-defined Functions: Special functions provided by Databricks for specific operations.

The location of a function depends on whether it is built-in or user-defined.

1.2 Identifying the location of Built-in functions

Databricks provides many built-in functions. These do not belong to a catalog or schema but are available globally.
To check if a built-in function exists, run the following command

```
DESCRIBE FUNCTION function_name;
```

This command provides details on the built-in function and does not show a specific location because built-in functions are globally accessible.

1.3 Identifying the location of User-Defined Functions (UDFs)

User-defined functions (UDFs) reside within designated catalogs and schemas. Their localization requires structured querying. In environments leveraging **Unity Catalog**, the system.information_schema.routines table provides a structured repository of function metadata, facilitating cross- catalog function discovery.

2

```
SELECT routine_catalog, routine_schema, routine_name FROM
system.information_schema.routines
WHERE routine_name = 'function_name'; If
```

the output comes as follows:

routine_catalog	routine_schema	routine_name
main_catalog	analytics	function_name

The function 'function_name' is stored in **main_catalog** under the **analytics** schema.This helps in identifying the precise location of the function for querying, governance, and management.

1.4 Best Practices

- Leverage Unity Catalog for function storage, ensuring controlled access and structured visibility.
- Adopt standardized naming conventions for functions, promoting consistency and maintainability.
- Maintain documentation detailing function locations, dependencies, and invocation methodologies.

2. Security Model for Sharing SQL UDF's

Sharing SQL User-Defined Functions (UDFs) within Databricks follows a robust security model to ensure controlled access and prevent unauthorized usage. Role-based access control (RBAC) allows users and groups to be assigned roles to control function access at the catalog, schema, or function level. Administrators can grant or revoke privileges using SQL commands such as

GRANT EXECUTE ON FUNCTION catalog_name.schema_name.function_name TO user_or_group.

Functions stored within Unity Catalog inherit the security policies applied at the catalog and schema levels, providing centralized visibility into who accessed and modified a function. UDFs executed in Databricks run in

isolated environments to prevent unauthorized data access, restricting network access and file I/O unless explicitly permitted.

Databricks Audit Logs track all function execution activities, enabling security teams to monitor usage patterns. Logs include user details, timestamps, and function call records. Organizations can implement data masking policies on UDF output to restrict sensitive data exposure, enforced using SQL security policies within Unity Catalog. By leveraging these security principles, organizations can confidently share and execute SQL UDFs while maintaining compliance, governance, and data integrity.

3. CASE/WHEN in SQL

In SQL, the CASE statement is a powerful conditional expression that allows for complex decision-making within queries. It is particularly useful for implementing conditional logic in SELECT, UPDATE, INSERT, and ORDER BY statements. The CASE statement evaluates conditions sequentially and returns the first matching result.

Case When:

- Returns resN for the first optN that equals expr or def if none matches.
- Returns resN for the first condN evaluating to true, or def if none found.

3.1 Syntax

CASE

```
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
        [……]
    ELSE default_result
END
```

3.2 Examples

```
SELECT employee_name, salary,
    CASE
```



```
        WHEN salary > 100000 THEN 'High Income'
        WHEN salary BETWEEN 50000 AND 100000 THEN 'Middle Income'
        WHEN salary < 50000 THEN 'Low Income' ELSE
        'Unknown'
    END AS income_category FROM
employees;
```

Employee_name	Salary	Income_category
John	120000	High Income
Smith	75000	Middle Income
Brown	45000	Low Income

```
SELECT product_name, category
FROM products
ORDER BY
    CASE category
        WHEN 'Electronics' THEN 1
        WHEN 'Furniture' THEN 2
        WHEN 'Clothing' THEN 3
        ELSE 4
    END;
```

product_name	Category
Electronics	1
Clothing	3
Food	4

4. CASE/WHEN for Custom Control Flow

The CASE statement in SQL is a powerful tool that facilitates the implementation of custom control flow logic within queries. It enables dynamic decision-making by evaluating conditions and returning corresponding results based on specific criteria. This functionality is widely used in SELECT, UPDATE, ORDER BY, and aggregation queries to manipulate data effectively. In scenarios where conditional processing is required, the CASE statement can be employed to streamline logic within a single query, reducing the need for multiple statements or procedural logic.

4.1 Examples

```
SELECT
   COUNT(CASE WHEN job_role = 'Manager' THEN 1 END) AS managers,
   COUNT(CASE WHEN job_role = 'Developer' THEN 1 END) AS developers,
   COUNT(CASE WHEN job_role = 'Analyst' THEN 1 END) AS analysts
FROM employees;
```

managers	developers	analysts
8	12	7

```
UPDATE employees SET
salary =
   CASE
      WHEN performance_rating = 'Excellent' THEN salary * 1.20
      WHEN performance_rating = 'Good' THEN salary * 1.10 ELSE
      salary * 1.05
   END;
```

Emp_id	Emp_name	Performance_rating	Salary
223	John	Excellent	
224	Mathew	Good	
225	Jane	Average	

```
SELECT ticket_id, issue_type, priority
FROM support_tickets
ORDER BY
   CASE priority
      WHEN 'High' THEN 1
      WHEN 'Medium' THEN 2
      WHEN 'Low' THEN 3
      ELSE 4
   END;
```

Before Sorting:

Ticket_id	Issue_type	Priority
101	Login Issue	Medium
102	Payment Fail	High
103	UI Bug	Low
104	Account Locked	High
105	Feature Request	Medium

After Sorting:

Ticket_id	Issue_type	Priority
102	Payment Failure	High
104	Account Locked	High
101	Login Issue	Medium
105	Feature Request	Medium
103	UI Bug	Low

4.2 Best Practices

- Keep It Simple and Readable
- Use ELSE to Manage Default Cases
- Use CASE Inside Aggregate Functions Wisely
- Use CASE in ORDER BY for Custom Sorting
- Avoid Redundant Conditions

1. Where Delta Lake Provides ACID Transactions

Delta Lake ensures ACID (Atomicity, Consistency, Isolation, Durability) transactions through its transaction log, which records all changes made to the data. This mechanism guarantees that each transaction is processed reliably, even in the event of system failures or concurrent operations.

2. Benefits of ACID Transactions

ACID transactions offer several advantages:

- **Data Integrity**: They ensure that data remains accurate and consistent, even in the face of system failures or concurrent access.
- **Consistency Across Systems**: ACID transactions maintain data validity, ensuring that only data adhering to predefined rules is written to the database.
- **Fault Tolerance**: They provide resilience by ensuring that committed transactions are durable, even if a system crash occurs immediately afterward.
- **Concurrency Control**: ACID transactions allow multiple operations to run concurrently without leading to inconsistent or partial results, thus enabling high throughput in multi- user environments.
- **Simplified Data Management**: They abstract away the complexities of ensuring consistency and isolation, allowing users to focus more on analyzing the data rather than worrying about transactional details.

3. How to Identify Whether a Transaction is ACID-Compliant

A transaction is ACID-compliant if it adheres to the following properties:

- **Atomicity**: The transaction is fully completed or fully rolled back; there are no partial commits.
- **Consistency**: The transaction brings the database from one valid state to another, maintaining data integrity.
- **Isolation**: Concurrent transactions do not interfere with each other, ensuring that intermediate states are not visible to other transactions.
- **Durability**: Once a transaction is committed, its changes are permanent, even in the event of a system crash.

Tools like Delta Lake can help manage and verify these properties, as their transaction logs enforce ACID guarantees.

4. Compare and Contrast Data and Metadata

Data and metadata serve different purposes:

- **Data**: This is the raw information stored in a database or data lake. It can be structured (e.g., rows in a table) or unstructured (e.g., images, text files).
    - *Example*: Customer names, product descriptions, or transaction records.
    - *Role*: Represents the core information users need to analyze or process.
- **Metadata**: This is data about data. It describes the characteristics, structure, and properties of the actual data.
    - *Example*: A file's size, creation date, or the schema describing how the data is organized (e.g., column names and types in a table).
    - *Role*: Provides context to the actual data, making it more understandable and usable.

Key Differences:

- **Role**: Data is the content being analyzed or processed, while metadata provides information about the data's structure, usage, or provenance.
- **Changeability**: Data changes based on user interactions or transactions, while
  metadata is typically static or changes infrequently (e.g., when data structure changes).

- **Example Use**: Data might represent sales transactions, while metadata would describe the column names, types (e.g., "Amount: float"), or how the data should be interpreted.

In summary, data is the raw content, and metadata provides essential information for managing, interpreting, and using that data effectively.

1. Compare and Contrast Managed and External Tables

Managed Table

A managed table (also called an internal table) is a table where Databricks manages both the metadata and the data.

Key Characteristics

- Data is stored in Databricks' default storage location.
- The table and its underlying data are deleted together when the table is dropped.
- Best suited for temporary or non-critical datasets that don't need external storage.

Storage Location

- Typically stored in DBFS (Databricks File System) under:
  - dbfs:/user/hive/warehouse/<database_name>.db/<table_name>
- The storage location is automatically managed and does not require manual specification.

Pros

- Easy to manage – Databricks handles everything.
- Good performance – Optimized for Spark queries.
- No external dependencies – Fully integrated within Databricks.

Cons

- Dropping the table deletes the data permanently.
- Not ideal for sharing with other platforms outside Databricks.

External Table

An external table (also called an unmanaged table) only stores metadata in Databricks, while the actual data resides in an external location like Azure Data Lake Storage (ADLS), Amazon S3, or a mounted DBFS location.

Key Characteristics

- The data remains intact even if the table is dropped.
- Requires explicitly specifying the storage location at table creation.
- Ideal for long-term storage, data sharing, and compliance use cases.

Storage Location

- The data can be stored anywhere, such as:
  - abfss://my-container@my-storage-account.dfs.core.windows.net/path/
  - s3://my-bucket/path/
  - dbfs:/mnt/external_data/

Pros

- Data remains even if the table is dropped.
- Can be accessed by multiple platforms (Databricks, Power BI, Snowflake, etc.).
- Best for governance and compliance, ensuring data persists across platforms.

Cons

- Performance can be slower if external storage is not optimized.
- Requires additional setup to define external storage locations.

Comparison Table

FEATURE	MANAGED TABLE	EXTERNAL TABLE
STORAGE LOCATION	DBFS (Databricks default storage)	External storage (ADLS, S3, DBFS mount, etc.)
DATA OWNERSHIP	Databricks manages data and metadata	User manages data, Databricks manages only metadata
DATA DELETION	Data is deleted when the table is dropped	Data remains even after dropping the table
USE CASE	Temporary data, fast query performance	Long-term storage, external data integration
PERFORMANCE	Optimized for Databricks	Depends on external storage performance

---

2. Identify a Scenario to Use an External Table

External tables are useful in real-world scenarios such as:

Scenario 1: Data Lake Integration

- A company stores raw data in Azure Data Lake Storage (ADLS).
- Instead of copying data into Databricks, an external table is created to allow querying without duplication.
- This approach is cost-effective and ensures data consistency across platforms.

Example

```
CREATE EXTERNAL TABLE sales_data (
    transaction_id STRING,
    product STRING,
    amount DOUBLE
)
USING DELTA
LOCATION 'abfss://my-container@my-storage-
account.dfs.core.windows.net/sales_data/';
```

Now, the table points to ADLS, and the data remains there.

Scenario 2: Data Sharing Between Platforms

- If a company wants to share data between Databricks and Power BI, using external tables in ADLS allows multiple tools to read the same data.

Example

- An external table in Databricks allows data to be queried in Power BI, reducing storage duplication.

Scenario 3: Compliance and Governance

- Organizations may need to preserve data for audits and avoid accidental deletion.

- External tables ensure data persists even if someone mistakenly drops the table in Databricks.

3. Create a Managed Table

A managed table is simple to create and is fully handled by Databricks.

Using SQL

```sql
CREATE TABLE employees (
    employee_id INT,
    employee_name STRING,
    department STRING,
    salary DOUBLE
);
```

- This table is stored in Databricks' default managed storage.

Inserting Data

```sql
INSERT INTO employees VALUES (1, 'John Doe', 'IT', 75000);
INSERT INTO employees VALUES (2, 'Jane Smith', 'HR', 65000);
```

Using PySpark

```python
from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder.appName("Databricks").getOrCreate()

# Create Sample Data
data = [(1, "John Doe", "IT", 75000),
        (2, "Jane Smith", "HR", 65000)]

df = spark.createDataFrame(data, ["employee_id", "employee_name",
"department", "salary"])

# Write Data as a Managed Table
df.write.mode("overwrite").saveAsTable("employees")
```

- The table is now available in Databricks' Hive Metastore.
- Dropping this table will delete both metadata and data.

---

4. Identify the Location of a Table

To check where a table is stored, you can use the **DESCRIBE EXTENDED** command.

Using SQL

```
DESCRIBE EXTENDED employees;
```

- This returns metadata, including the storage location.
- If it's a managed table, the location will be:
  - dbfs:/user/hive/warehouse/employees/
- If it's an external table, the location will point to the external storage path.

Using PySpark

```
spark.sql("DESCRIBE EXTENDED employees").show(truncate=False)
```

- The output will include:
  - Type (Managed or External)
  - Storage Location
  - Schema Details

Inspect the Directory Structure of Delta Lake Files

Delta Lake organizes data into a directory structure that includes both data files and a transaction log. The transaction log is crucial for Delta Lake's ACID (Atomicity, Consistency,
Isolation, Durability) properties and is stored in the _delta_log directory. This directory contains JSON files, each representing a version of the table. These JSON files record every change made to the table, such as additions, deletions, and updates.

- **Data Files**: These are stored in Parquet format and contain the actual data.
- **Transaction Log**: Located in the _delta_log directory, it includes:

- **Checkpoint Files**: These are Parquet files that provide a snapshot of the table's state at a particular version.
- **Delta Files**: JSON files that record individual changes to the table.

Identify Who Has Written Previous Versions of a Table

To identify who has written previous versions of a Delta table, you can use the DESCRIBE HISTORY command. This command provides a comprehensive history of all operations performed on the table, including details about the user who performed each operation.

Example:

DESCRIBE HISTORY

table_name; The output

includes:

- **Version**: The version number of the table.
- **Timestamp**: The time when the operation was performed.
- **User**: The user who performed the operation.
- **Operation**: The type of operation (e.g., WRITE, DELETE, MERGE).
- **Operation Parameters**: Additional details about the operation.

Review a History of Table Transactions

The DESCRIBE HISTORY command also allows you to review the history of transactions on a Delta table. This command provides a detailed log of all operations, making it easy to track changes and understand the table's evolution over time.

Example:

DESCRIBE HISTORY

table_name LIMIT 1; The

output includes:

- **Version**: The version number of the table.
- **Timestamp**: The time when the operation was performed.

- **User**: The user who performed the operation.
- **Operation**: The type of operation (e.g., WRITE, DELETE, MERGE).
- **Operation Parameters**: Additional details about the operation.
- **Cluster ID**: The ID of the cluster where the operation was performed.
- **Read Version**: The version of the table that was read during the operation.
- **Isolation Level**: The isolation level used during the operation.

Roll Back a Table to a Previous Version

Delta Lake allows you to roll back a table to a previous version using the RESTORE command. This command can be used to revert the table to a specific version or timestamp, effectively undoing any changes made after that point.

Example:

RESTORE TABLE table_name TO VERSION AS OF 1;

You can also restore to a specific timestamp:

RESTORE TABLE table_name TO TIMESTAMP AS OF '2023-01-01T00:00:00Z';

The RESTORE command is useful for:

- **Undoing Mistakes**: Reverting accidental changes.
- **Data Recovery**: Recovering data from a previous state.
- **Testing**: Rolling back to a known state for testing purposes.

1. Rolling Back a Table to a Previous Version

Delta Lake provides the ability to time-travel, which means you can roll back a table to a
previous version. This is particularly useful in scenarios where you need to revert to a prior state of the data due to errors, accidental changes, or for auditing purposes.

- How to Roll Back a Table:

To roll back a table to a previous version, you can use the `RESTORE` command. This command allows you to revert the

table to a specific version or timestamp.

- sql code

  RESTORE TABLE my_table TO VERSION AS OF 5;

In this example, `my_table` is rolled back to version 5. You can also use a timestamp to restore the table to a specific point in time:

- sql code

  RESTORE TABLE my_table TO TIMESTAMP AS OF '2023-10-01T00:00:00Z';

Key Points:

- **Auditability**: Rolling back to a previous version is a powerful feature for auditing and debugging.
- **Data Integrity**: Ensures that you can recover from accidental changes or corrupt data.
- **Versioning**: Delta Lake maintains a transaction log that tracks every change made to the table, enabling precise rollbacks.

2. Querying a Specific Version of a Table

Delta Lake's time-travel feature also allows you to query a specific version of a table without rolling it back. This is useful for historical analysis or comparing data across different versions.

- How to Query a Specific Version:

You can query a specific version of a table using the `VERSION AS OF` or `TIMESTAMP AS OF` syntax in your SQL query.

- sql code

```
SELECT * FROM my_table VERSION AS OF 5;
```

Or using a timestamp:

- sql code

```
SELECT * FROM my_table TIMESTAMP AS OF '2023-10-01T00:00:00Z';
```

Key Points:

- **Historical Analysis**: Enables you to analyze data as it existed at a specific point in time.
- **No Impact on Current Data**: Querying a specific version does not alter the current state of the table.
- **Compliance**: Useful for regulatory compliance where historical data access is required.

3. Why Z-Ordering is Beneficial to Delta Lake Tables

Z-ordering is a technique used to optimize the layout of data in Delta Lake tables, particularly for large datasets. It improves query performance by colocating related information in the same set of files, which reduces the amount of data read during queries.

-> Benefits of Z-Ordering:

- **Improved Query Performance:** By colocating related data, Z-ordering reduces the amount of data scanned during queries, leading to faster query execution.
- **Optimized Data Skipping**: Z-ordering enhances Delta Lake's data skipping capabilities, which means that fewer files need to be read to satisfy a query.
- **Efficient Storage**: Reduces the overall storage footprint by minimizing the number of files that need to be accessed.

- How to Apply Z-Ordering:

You can apply Z-ordering using the `OPTIMIZE` command and specifying the columns to Z- order:

- sql code

  OPTIMIZE my_table ZORDER BY (column1, column2);

Key Points:

- **Column Selection**: Choose columns that are frequently used in query predicates (e.g.,
`WHERE` clauses) for Z-ordering.

- **Resource Intensive**: Z-ordering can be resource-intensive, so it's best applied during off-peak hours or as part of a scheduled maintenance job.

- **Incremental Optimization**: Delta Lake's `OPTIMIZE` command is incremental, meaning it only processes new or changed data, making it efficient for ongoing optimization.

4. How VACUUM Commits Deletes

The `VACUUM` command in Delta Lake is used to physically delete files that are no longer referenced by the table's transaction log. This is important for managing storage and ensuring that obsolete data is removed.

-> How VACUUM Works:

- Logical Deletes: When you delete data from a Delta Lake table, the data is not immediately removed from storage. Instead, Delta Lake marks the data as deleted in the transaction log.

- Physical Deletes: The `VACUUM` command physically removes the files that are no longer needed, freeing up storage space.

-> Example Usage:

To run the `VACUUM` command, you can specify a retention period (in hours) to determine which files should be deleted:

- sql code

VACUUM my_table RETAIN 168 HOURS;

This command deletes files that are older than 168 hours (7 days) and are no longer referenced by the table.

Key Points:

- **Data Retention**: Be cautious with the retention period, as setting it too low can result in the loss of historical data needed for time-travel.

- **Irreversible Operation**: Once `VACUUM` is executed, the deleted files cannot be recovered, so ensure that you have proper backups if needed.

- **Storage Management**: Regularly running `VACUUM` helps manage storage costs, especially in environments with frequent data updates or deletions.

Identify the kind of files Optimize compacts.

In **Databricks**, the **Optimize** command is used to compact **small files** in **Delta Lake** tables. These small files are typically created due to frequent writes, updates, and inserts, leading to inefficient storage and query performance.

Types of Files Optimized by Optimize Command

1. **Delta Lake Small Files**: Files that are created when data is ingested incrementally, resulting in many small parquet files.
2. **Parquet Files in Delta Tables**: The Optimize command compacts these files into larger ones to improve query performance.
3. **Data Skewed Files**: Unevenly distributed small files that can cause performance issues.

How It Works

- The **Optimize** command merges these small files into **larger parquet files**, reducing metadata overhead.
- It improves **query performance** by reducing the number of files scanned.

Example Command

```
OPTIMIZE my_table [FULL] [WHERE predicate]
```

For further efficiency, **ZORDER BY** can be used to optimize

queries with filtering: OPTIMIZE my_table

ZORDER BY (column_name)

Identify CTAS as a solution.

CTAS (Create Table As Select) is a powerful technique in **Databricks** that helps optimize performance and manage **small file issues** in Delta Lake tables. It can be used as an alternative or complement to **OPTIMIZE**.

How CTAS Helps with Small Files

1. **Creates a New Optimized Table**: Instead of modifying the existing table, CTAS creates a new table with optimized storage layout.
2. **Compacts Small Files**: When data is written to the new table, **Databricks automatically optimizes** file sizes.
3. **Improves Query Performance**: Reduces metadata overhead and **improves Spark SQL query efficiency**.

CTAS Syntax

```
CREATE TABLE optimized_table USING DELTA
AS
SELECT * FROM original_table;
```

- This creates a **new Delta table** with optimized data storage.
- It automatically compacts small files into **fewer, larger Parquet files**.
- Can be used when **Optimize** is not reducing file fragmentation effectively.

When to Use CTAS vs. OPTIMIZE

Feature	**CTAS**	**OPTIMIZE**
Use Case	Creating a new optimized table	Compacts small files in an existing table
Performance	Better for large-scale compaction	Works well for moderate file compaction
Metadata Handling	Reduces old table's metadata load	Does not alter metadata significantly

Create a generated column.

In **Delta Lake**, a **generated column** is a special type of column whose values are automatically computed based on a user-defined expression involving other columns in the same table. This feature ensures data consistency and can enhance query performance by precomputing values that would otherwise require computation at query time.

Creating a Table with Generated Columns

To define a table with a generated column, you specify the GENERATED ALWAYS AS clause followed by the desired expression. For example, to create a table where the dateOfBirth column is automatically derived from the birthDate column:

```
CREATE TABLE
 default.people10m ( id
 INT,
 firstName
 STRING,
 middleName
 STRING,
 lastName
 STRING, gender
 STRING,
 birthDate
 TIMESTAMP,
 dateOfBirth DATE GENERATED ALWAYS AS
 (CAST(birthDate AS DATE)), ssn STRING,
 salary INT
)
```

In this schema, dateOfBirth is a generated column that computes its value by casting the birthDate timestamp to a date.

Considerations and Limitations

- **Storage**: Generated columns are physically stored in the table, consuming storage space similar to regular columns.
- **Expression Restrictions**: The generation expression can utilize

deterministic SQL functions but cannot include:

  - User-defined functions
  - Aggregate functions
  - Window functions
  - Functions returning multiple rows

- **Partitioning**: Defining partition columns using specific expressions on generated columns can optimize query performance. For instance, partitioning by YEAR(birthDate) or DATE_FORMAT(birthDate, 'yyyy-MM') is supported and can enhance data retrieval efficiency.

Example: Partitioning with Generated Columns

To partition a table by year and month based on a timestamp column:

```
CREATE TABLE events (
 eventId STRING,
 eventType
 STRING,
 eventTimestamp TIMESTAMP,
 eventDate DATE GENERATED ALWAYS AS (CAST(eventTimestamp
 AS DATE))
)
PARTITIONED BY (YEAR(eventTimestamp), MONTH(eventTimestamp))
```

In this setup, the table is partitioned by the year and month extracted from the eventTimestamp, facilitating efficient query performance for time-based data retrieval.

Add a table comment.

In **Azure Databricks**, you can add comments to tables to provide metadata and documentation. This helps improve readability, maintainability, and collaboration.

1. **Adding a Comment While Creating a Table**

You can specify a table comment using the COMMENT clause in the CREATE TABLE statement:

```
CREATE TABLE
  customer_data (
  customer_id STRING,
  name
  STRING,
  age INT
) USING DELTA
COMMENT 'This table stores customer details including ID, name, and age.';
```

2. Adding a Comment to an Existing Table

If a table is already created, you can add or update the comment

using ALTER TABLE: ALTER TABLE customer_data
SET TBLPROPERTIES ('comment' = 'Updated table storing customer details.');

3. Viewing the Table Comment

To check the table's comment, use:

DESCRIBE TABLE customer_data;

This will display metadata, including the comment.

1. Use CREATE OR REPLACE TABLE and INSERT OVERWRITE

CREATE OR REPLACE TABLE: This command is used to create a new table or replace an existing table with the same name. It ensures that the table is created with the
specified schema and properties. If the table already exists, it is dropped and recreated with the new definition.

Syntax:

```
CREATE OR REPLACE TABLE
  table_name ( column1 datatype,
  column2 datatype,
  ...
);
```

Example:

```
CREATE OR REPLACE TABLE
```

```
  employees ( id INT,
  name
  STRING,
  age INT,
  department STRING
);
```

INSERT OVERWRITE: This command is used to overwrite the existing data in a table or partition with new data. It removes all existing data and inserts the new data specified in the query. This is useful when you need to refresh the entire dataset in a table or partition.

Syntax:

```
INSERT OVERWRITE TABLE
table_name SELECT columns
FROM source_table
WHERE
condition;
```

Example:

```
INSERT OVERWRITE TABLE employees
SELECT id, name, age,
department FROM
new_employees
WHERE department = 'Sales';
```

2. Compare and Contrast CREATE OR REPLACE TABLE and INSERT OVERWRITE

- **Purpose**:
  - **CREATE OR REPLACE TABLE**: Used to create or redefine the structure of a table.
  - **INSERT OVERWRITE**: Used to replace the data within an existing table or partition.
- **Operation**:
  - **CREATE OR REPLACE TABLE**: Drops the existing table (if any) and creates a new one with the specified schema.
  - **INSERT OVERWRITE**: Deletes the existing data in the table or partition and inserts new data.
- **Use Case**:
  - **CREATE OR REPLACE TABLE**: When you need to change the schema or properties of a table.
  - **INSERT OVERWRITE**: When you need to update the data in a table without changing its schema.

3. Identify a Scenario in Which MERGE Should Be Used

Scenario: Use the **MERGE** command when you need to perform complex operations such as updating, inserting, or deleting records in a table based on the results of a join with another table. For example, when synchronizing data between two tables where you need to update existing records, insert new records, and delete obsolete records in a single operation.

Syntax:

MERGE INTO target_table AS target

```
  USING source_table AS
  source ON target.id =
  source.id WHEN
  MATCHED THEN
    UPDATE SET target.name =
        source.name, target.age =
        source.age
  WHEN NOT MATCHED THEN
INSERT (id, name,
age)
    VALUES (source.id, source.name,
  source.age) WHEN NOT MATCHED
  BY SOURCE THEN
    DELETE;
```

Example:

```
  MERGE INTO employees
  AS target USING
  new_employees AS source
  ON target.id = source.id
WHEN MATCHED
THEN
    UPDATE SET target.name =
        source.name, target.age =
        source.age
  WHEN NOT MATCHED THEN
INSERT (id, name,
age)
    VALUES (source.id, source.name,
```

```
source.age)  WHEN  NOT  MATCHED
BY SOURCE THEN
  DELETE;
```

4. Identify MERGE as a Command to Deduplicate Data Upon Writing

MERGE: The **MERGE** command can be used to deduplicate data upon writing by
specifying conditions to match records between the source and target tables. When duplicates are found, you can define actions to update or delete the existing records, ensuring that only unique records are retained in the target table.

Example:

```
MERGE  INTO  employees
AS  target  USING
new_employees  AS  source
ON target.id = source.id
WHEN MATCHED THEN
  UPDATE  SET  target.name  =
      source.name,  target.age  =
      source.age
WHEN NOT MATCHED THEN
  INSERT (id, name, age)
  VALUES  (source.id,  source.name,
source.age)  WHEN  NOT  MATCHED
BY SOURCE THEN
  DELETE;
```

1. Benefits of the MERGE Command

The MERGE command in SQL is a powerful tool that allows you to perform INSERT, UPDATE, and DELETE operations in a single atomic statement. Here are some benefits:

- **Atomic Operations**: Combines multiple operations into one, ensuring data consistency.
- **Efficiency**: Reduces the complexity of coding by handling multiple operations in a single statement.
- **Concurrency Control**: Helps in managing concurrent data modifications effectively.
- **Flexibility**: Allows conditional logic to determine whether to insert, update, or delete records.

Syntax:

```
sql

MERGE INTO target_table AS

target USING source_table AS

source

ON target.id =

source.id WHEN

MATCHED THEN

  UPDATE SET target.column1 =

source.column1 WHEN NOT

MATCHED THEN

  INSERT (column1, column2) VALUES (source.column1, source.column2);
```

Example:

```
sql

MERGE INTO employees

AS target USING

new_employees AS source

ON target.employee_id =
```

source.employee_id WHEN

MATCHED THEN

UPDATE SET target.salary =

source.salary WHEN NOT

MATCHED THEN

INSERT (employee_id, name, salary) VALUES (source.employee_id, source.name, source.salary);

2. Why a COPY INTO Statement is Not Duplicating Data

The COPY INTO statement in Databricks is designed to be idempotent, meaning it will not load the same file more than once. This ensures that data is not duplicated in the target table. If files have already been loaded, they are skipped even if they have been modified since they were loaded.

3. Scenario in Which COPY INTO Should Be Used

The COPY INTO command is ideal for batch data loading from external storage into a Delta table. It is particularly useful when you need to load large volumes of data efficiently and ensure that files are not duplicated.

4. Using COPY INTO to Insert Data

The COPY INTO command can be used to load data from files into a Delta table. Here is the syntax and an example:

Syntax:

sql

COPY INTO

target_table FROM

'file_path'

FILEFORMAT =

file_format [OPTIONS];

Example:

sql

COPY INTO

delta.`/mnt/delta/target_table`

FROM 'dbfs:/mnt/source_data/'

FILEFORMAT = PARQUET;

Identify the components necessary to create a new DLT pipeline.

Necessary Components:

1. Cluster creation permission or access to a cluster policy.
2. Serverless compute (optional but recommended).
3. Unity Catalog Schema creation or access with appropriate privileges.
4. Volume creation or access with appropriate privileges.

Explanation

To create a new **Delta Live Tables (DLT) pipeline**, the following components and permissions are necessary:

1. **Cluster Permission**
    - You must have **cluster creation permission** or access to a **cluster policy**

defining a **Delta Live Tables cluster**.

    - The DLT runtime automatically creates a cluster before running the pipeline, and it will fail if you lack the required permissions.
2. **Serverless Pipeline (Optional)**
    - All users can trigger updates using **serverless pipelines** by default.
    - **Serverless must be enabled** at the account level and might not be available in all regions.
3. **Schema for Unity Catalog**
    - Databricks recommends creating a **new schema** to run DLT pipelines.
    - Required permissions to create a schema:
        - **ALL PRIVILEGES** or
        - **USE CATALOG** and **CREATE SCHEMA** privileges.
    - If using an **existing schema**, the necessary permissions are:

- **USE CATALOG** for the parent catalog.
- **ALL PRIVILEGES** or
- **USE SCHEMA**, **CREATE MATERIALIZED VIEW**, and **CREATE TABLE** privileges.

4. **Volume for Storing Data (Optional)**
   - Recommended to create a **new volume** for storing sample data.
   - Required permissions to create a volume in an **existing schema**:
     - **USE CATALOG** for the parent catalog.
     - **ALL PRIVILEGES** or
     - **USE SCHEMA** and **CREATE VOLUME** privileges.
   - If using an **existing volume**, the necessary permissions are:
     - **USE CATALOG** for the parent catalog.
     - **USE SCHEMA** for the parent schema.
     - **ALL PRIVILEGES** or **READ VOLUME** and **WRITE VOLUME** on the target volume.

These components and permissions ensure a successful **DLT pipeline** setup in Databricks.

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

Target

In a Databricks pipeline, the "**target**" refers to the final destination where processed data is written, essentially the table or location within your data lake where the transformed data will be stored after going through the pipeline steps; it defines the target schema and location where the output of your pipeline will be saved.

Notebook Libraries

Notebook libraries in Databricks allow users to create custom environments for specific notebooks. These environments can be used for Python, R, and other languages.

When creating a data pipeline in Databricks using notebooks, key libraries to utilize include: "pyspark" for core Spark DataFrame manipulation, "dbutils" for Databricks utility functions, "pandas" for data manipulation in

a more familiar Python style, "koalas" for large-scale pandas operations on Spark dataframes, and relevant libraries specific to your data source (like "jdbc" for connecting to databases) depending on your pipeline needs, you may also use libraries like "mllib" for machine learning tasks, "delta-lake" for managing Delta Lake tables, and libraries for data visualization like "matplotlib" or "plotly" within your notebooks.

Notebook-scoped libraries: Specific to a notebook and its runtime environment. These libraries are not persistent and must be reinstalled for each session.

Installing libraries

- To install notebook-scoped libraries, use the %pip or %conda commands for Python libraries.

Using notebook libraries

Notebook libraries can be used to create, modify, save, reuse, and share custom environments. They can also be used to develop code and present results.

When to use notebook libraries

Use notebook-scoped libraries when you need a custom environment for a specific notebook

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

Aspect	Triggered Pipeline	Continuous Pipeline
Cost	Lower cost because the cluster runs only long enough to complete the update. Resources are used only when triggered.	Higher cost as the cluster must run continuously, consuming resources even when no new data arrives.
Latency	Higher latency since new data is processed only when the pipeline is triggered (e.g., every 10 minutes, hourly, or daily).	Lower latency as data is processed as soon as it arrives, keeping tables updated in near real-time (between 10 seconds and a few minutes).

Summary:

- **Triggered pipelines** are cost-effective but introduce delays in data freshness since updates only occur at scheduled intervals. New data won't be processed until the pipeline is triggered.
- **Continuous pipelines** provide real-time updates with minimal latency but incur higher costs due to the always-running cluster.

Identify which source location is utilizing Auto Loader.

Supported Auto Loader sources.

Auto Loader can load data files from the following sources:

- Amazon S3 (s3://)
- Azure Data Lake Storage Gen2 (ADLS Gen2, abfss://)
- Google Cloud Storage (GCS, gs://)
- Azure Blob Storage (wasbs://)
- ADLS Gen1 (adl://)
- Databricks File System (DBFS, dbfs:/).

Auto Loader can ingest JSON, CSV, XML, PARQUET, AVRO, ORC, TEXT, and BINARYFILE file formats.

DATABRICKS AUTO LOADER: KEY SCENARIOS AND BEHAVIORS

1. Scenario Where Auto Loader is Beneficial

Use Case: Real-time Traffic Data Ingestion for Smart City Management

Auto Loader in Databricks is particularly useful in scenarios where data is continuously arriving in a cloud storage system (e.g., Azure Blob Storage, AWS S3) and needs to be incrementally ingested into a Delta Lake or other storage formats without manual intervention.

Example Scenario: A smart city project is using Auto Loader to monitor real-time traffic data. IoT sensors and cameras placed at various intersections continuously record vehicle counts, speeds, and traffic flow patterns. This data needs to be ingested into Databricks for real-time analysis, traffic optimization, and incident detection.

How Auto Loader Helps:

- **Continuous Data Ingestion:** Auto Loader detects new traffic data

files and processes them incrementally.

- **Schema Evolution Handling:** As new sensor types or additional data points (e.g.,
  vehicle type, weather conditions) are added, Auto Loader can handle schema changes dynamically.
- **Low Latency s Scalability:** Ensures minimal delay in ingestion and processing, helping detect traffic congestion and incidents in real time.
- **Real-Time Alerts:** Enables real-time traffic management, such as adjusting traffic signal timings or dispatching emergency services to accident sites.

This automation enhances traffic flow, reduces congestion, and improves overall city management while minimizing manual intervention.

2. Why Auto Loader Infers All Data as STRING from a JSON Source

When Auto Loader ingests JSON data, all fields may be inferred as STRING instead of their actual data types. This is due to the way JSON stores data without explicit type definitions.

Possible Reasons and Solutions:

1. **Default Behavior of Schema Inference:**
   - Auto Loader defaults to treating all columns as STRING because JSON files do not inherently encode data types.

Solution: Enable column type inference using cloudFiles.inferColumnTypes. df = (spark.readStream

```
.format("cloudFiles")
.option("cloudFiles.format", "json")
.option("cloudFiles.inferColumnTypes", "true")
.load("s3://path-to-data/"))
```

2. **Presence of Nested JSON Data:**

If a JSON file contains nested structures (arrays, objects), Auto Loader may treat them as STRING.

Solution: Define an explicit schema to handle nested data correctly.

```
from pyspark.sql.types import StructType, StructField, IntegerType,
```

StringType

```
schema = StructType([
  StructField("sensor_id", StringType(), True),
  StructField("temperature", IntegerType(), True),
  StructField("humidity", IntegerType(), True)
])

df = (spark.readStream
    .format("cloudFiles")
    .option("cloudFiles.format", "json")
    .schema(schema)
    .load("s3://path-to-data/"))
```

3. **Schema Evolution Issues:**
    - When new JSON files introduce new data types, Auto Loader defaults to STRING to prevent breaking changes.

Solution: Enable schema evolution.

```
df = (spark.readStream
    .format("cloudFiles")
    .option("cloudFiles.format", "json")
    .option("cloudFiles.schemaEvolutionMode", "addNewColumns")
    .load("s3://path-to-data/"))
```

For more details, refer to the Databricks documentation on schema inference and evolution.

3. Default Behavior of a Constraint Violation

What Happens When a Constraint is Violated? When a constraint violation occurs in Auto Loader or Delta Lake, the transaction fails with an error. This failure prevents incorrect or inconsistent data from being committed to the dataset.

Example: If a table has a constraint that requires temperature values to be

between 0 and 100 degrees, but an incoming record has a temperature of -5 degrees, the transaction will fail with an error message.

ALTER TABLE sensor_data ADD CONSTRAINT temperature_check CHECK (temperature BETWEEN 0 AND 100);

If an invalid record (temperature = -5) is ingested, the transaction fails, ensuring data integrity.

4. **Impact of ON VIOLATION DROP ROW and ON VIOLATION FAIL UPDATE**

When handling constraint violations, Databricks offers two key options: ON VIOLATION DROP ROW and ON VIOLATION FAIL UPDATE.

1. **ON VIOLATION DROP ROW**

Behavior:

- Invalid records are dropped before they are written to the target table.
- The count of dropped records is logged for monitoring.

Use Case: Useful when you want to automatically discard bad data while continuing the ingestion process.

Example:

COPY INTO sensor_data

FROM 's3://sensor-

bucket/' FILEFORMAT =

JSON

ON VIOLATION DROP ROW;

If any row contains invalid temperature values, it is dropped from ingestion.

2. **ON VIOLATION FAIL UPDATE**

Behavior:

- Invalid records prevent the update from succeeding.
- Manual intervention is required before reprocessing.
- Other independent processes in the pipeline are not affected.

Use Case: Used when data integrity is critical and manual review is required before ingestion.

Example:

```
COPY INTO sensor_data
FROM 's3://sensor-
bucket/' FILEFORMAT =
JSON
ON VIOLATION FAIL UPDATE;
```

If a row contains an invalid temperature value, the entire transaction fails, ensuring no bad data is written.

1. **Change Data Capture (CDC) and Behavior of APPLY CHANGES INTO**

Change Data Capture (CDC) is a data integration technique that tracks changes in a source system, such as inserts, updates, and deletes, enabling efficient data synchronization. Instead of processing entire datasets repeatedly, CDC ensures only the changes are processed, reducing resource consumption and improving performance.

In Databricks, the **APPLY CHANGES INTO** statement simplifies CDC workflows within Delta Live Tables (DLT) pipelines. Unlike MERGE INTO, which can be complex due to out-of-sequence records and multiple updates to the same row, APPLY CHANGES INTO ensures proper ordering of updates and manages multiple modifications efficiently.

Syntax Example:

```
APPLY CHANGES INTO
live.target FROM
stream(cdc_data.users)
KEYS (userId)
APPLY AS DELETE WHEN operation
= "DELETE" APPLY AS TRUNCATE
WHEN operation = "TRUNCATE"
SEQUENCE BY sequenceNum
COLUMNS * EXCEPT (operation,
sequenceNum) STORED AS SCD
```

TYPE 2;

Key Components:

- FROM: Specifies the source of change data.
- KEYS: Defines primary keys for matching records.
- APPLY AS DELETE/TRUNCATE: Handles delete or truncate operations.
- SEQUENCE BY: Ensures events are processed in order.
- COLUMNS: Allows selective column updates while excluding certain fields.
- STORED AS SCD TYPE 2: Maintains historical data for tracking changes.

Types of Slowly Changing Dimensions (SCD) Supported:

- **SCD Type 1**: Overwrites existing records without history tracking.
- **SCD Type 2**: Maintains historical versions of records by adding new rows.
- **SCD Type 3**: Stores limited history by using additional columns for previous values.

Advantages of APPLY CHANGES INTO over MERGE INTO:

- Handles out-of-order event processing automatically.
- Reduces complexity in managing multiple update scenarios.
- Improves performance by efficiently handling incremental changes.

---

2. Querying the Events Log for Metrics, Audit Logging, and Lineage Examination

Databricks maintains an event log that records operational and transformation details within Delta Live Tables (DLT) pipelines. It is useful for monitoring pipeline performance,
troubleshooting issues, and tracking data lineage.

Querying Lineage Information:

To track data flow between input and output datasets:

```
SELECT
  details:flow_definition.output_dataset AS output_dataset,
  details:flow_definition.input_datasets AS input_dataset
```

```
FROM event_log_raw,
  latest_update
WHERE event_type =
'flow_definition' AND
origin.update_id =
latest_update.id;
```

Audit Logging for User Actions:

To log user actions within a DLT pipeline:

```
SELECT timestamp,
   details:user_action:action
   ,
   details:user_action:user_
name FROM
event_log_raw
WHERE event_type = 'user_action';
```

Retrieving Pipeline Execution Metrics:

To analyze DLT pipeline performance, including record counts and execution time:

```
SELECT timestamp,
   details:flow_progress.metrics.num_output_rows AS output_rows,
   details:flow_progress.metrics.execution_time_ms AS
   execution_time
FROM event_log_raw
WHERE event_type = 'flow_progress';
```

These queries help track modifications, user interactions, overall data flow, and performance metrics for auditing and debugging.

---

3. Troubleshooting DLT Syntax

When working with DLT pipelines, identifying syntax issues or errors is critical. The event log can help determine the specific notebook that caused an error.

Identifying the Notebook Producing Errors:

```
SELECT timestamp,
  details:error.message,
  details:error.stack_trace
FROM event_log_raw
WHERE event_type =
'error';
```

This query provides error messages and stack traces for debugging, helping users pinpoint the problematic transformation or function.

Need for LIVE in CREATE Statements:

When defining tables in Delta Live Tables, use LIVE to indicate that a table is continuously updated and managed by the DLT pipeline.

Example Without LIVE (Incorrect):

```
CREATE TABLE
customer_data AS SELECT *
FROM raw_data.customers;
```

Example With LIVE (Correct):

```
CREATE LIVE TABLE
customer_data AS SELECT *
FROM raw_data.customers;
```

Using LIVE ensures that customer_data stays synchronized with raw_data.customers, and the pipeline correctly tracks lineage and dependencies.

---

4. Identifying the Need for STREAM in FROM Clause

In DLT, using STREAM in the FROM clause ensures that data is processed as a continuous stream rather than as static snapshots. This is crucial for real-time updates and processing incremental changes efficiently.

Why Use STREAM?

- Enables continuous ingestion of new data instead of reprocessing the full dataset.
- Ensures real-time updates in downstream tables.
- Improves performance by only processing new records rather than scanning the entire table.

Example Without STREAM (Batch Processing):

SELECT * FROM raw_data.orders;

- Processes the full dataset, not just new records.
- Can lead to inefficient reprocessing of unchanged data.

Example With STREAM (Streaming Processing):

SELECT * FROM STREAM(raw_data.orders);

- Processes only newly arrived records.
- Avoids redundant computations and speeds up processing.

Use Cases for STREAM in DLT Pipelines:

- Capturing incremental updates from a CDC-enabled table.
- Processing real-time order transactions in an e-commerce application.
- Streaming sensor data from IoT devices.

1.

2. Identify the benefits of using multiple tasks in Jobs.

Benefits of Using Multiple Tasks in Databricks Jobs

1. Introduction

Databricks Jobs provide a way to orchestrate and execute workloads efficiently. A **multi-task**
job enables users to define and manage multiple tasks within a single job, optimizing workflow execution and improving resource management. Instead of running multiple independent jobs, using multiple tasks within a job ensures seamless execution, dependency management, and efficient parallelization.

What exactly is a job?

In Databricks, a job is used to schedule and orchestrate tasks on Databricks in a *workffow*. Common data processing workflows include ETL workflows,

running notebooks, and machine learning (ML) workflows, as well as integrating with external systems like dbt.

Jobs consist of one or more tasks and support custom control flow logic like branching (if / else statements) or looping (for each statement) using a visual authoring UI. Tasks can load or transform data in an ETL workflow, or build, train and deploy ML models in a controlled and repeatable way as part of your machine learning pipelines.

Databricks Jobs now supports task orchestration in public preview -- the ability to run multiple tasks as a directed acyclic graph (DAG). A job is a non-interactive way to run an application in a Databricks cluster, for example, an ETL job or data analysis task you want to run immediately or on a scheduled basis. The ability to orchestrate multiple tasks in a job significantly simplifies creation, management and monitoring of your data and machine learning workflows at no additional cost.

2. Key Benefits of Using Multiple Tasks

2.1 Modular Workflow Execution

- Instead of executing a monolithic script, workloads can be divided into multiple tasks.
- Each task can focus on a specific operation, making it easier to manage and debug.
- Enhances code reusability, as individual tasks can be used in multiple workflows.

2.2 Parallel Processing for Faster Execution

- Independent tasks can run in parallel, reducing overall execution time.
- Optimizes the use of compute resources by executing multiple steps simultaneously.

2.3 Dependency Management and Orchestration

- Users can set dependencies between tasks, ensuring a logical execution order.
- Provides flexibility to run specific steps sequentially or in parallel as needed.

2.4 Optimized Resource Allocation

- Each task can be assigned to a different compute cluster, optimizing resource usage.

- Prevents overloading a single cluster by distributing workloads efficiently.

2.5 Error Isolation and Automatic Retries

- If a single task fails, it can be retried independently without restarting the entire job.
- Reduces downtime by isolating errors to specific tasks instead of affecting the whole workflow.

2.6 Improved Observability and Debugging

- Logs and execution details are available for each task, making debugging easier.
- The Databricks UI provides a visual representation of the workflow, helping track execution progress.

3. Business-Oriented Example: E-Commerce Data Pipeline Scenario: Optimizing an E-commerce Data Processing Pipeline

A retail e-commerce company collects data from multiple sources, including website transactions, customer interactions, and inventory updates. They need to process this data efficiently to generate reports and update their analytics dashboards. Instead of running separate jobs for each step, they can create a **multi-task Databricks job** to streamline execution.

Multi-Task Job Workflow

Task Name	Description	Dependency
Extract Sales Data	Fetch sales data from transactional databases	None
Extract Inventory Data	Fetch latest inventory updates	None
Clean Sales Data	Perform data cleansing on sales records	Extract Sales Data
Clean Inventory Data	Standardize inventory records	Extract Inventory Data
Aggregate Sales Metrics	Compute total sales, revenue per category	Clean Sales Data

Aggregate Inventory Metrics	Compute stock levels, reorder points	Clean Inventory Data
Generate Business Reports	Merge metrics and create reports	Aggregate Sales Metrics, Aggregate Inventory Metrics

Key Benefits Observed

1. **Faster Processing** – Extract, clean, and aggregate steps run in parallel, reducing pipeline execution time.
2. **Efficient Resource Utilization** – Tasks use different compute clusters, optimizing cost and speed.
3. **Better Error Handling** – If data cleaning fails for sales data, it does not affect inventory processing.
4. **Scalability** – As data sources grow, new tasks can be added without modifying the entire pipeline.

4. Conclusion

Using multiple tasks in Databricks Jobs enables efficient workflow execution, better resource management, and streamlined debugging. Businesses dealing with large-scale data pipelines benefit from improved speed, error isolation, and modular workflow design, ensuring smooth data processing and analytics operations.

3. Set up a predecessor task in Jobs.

Setting Up a Predecessor Task in Databricks Jobs

1. Introduction

In Databricks Jobs, a **predecessor task** refers to a task that must be completed before another task can execute. This setup ensures proper workflow orchestration, enforces execution dependencies, and improves job efficiency.

2. Why Use Predecessor Tasks?

Predecessor tasks help manage job execution order efficiently. Benefits include:

- **Dependency Enforcement** – Ensures tasks execute in the correct sequence.
- **Optimized Workflows** – Prevents unnecessary waiting times by executing dependent tasks immediately after predecessors finish.

- **Fault Isolation** – If a predecessor task fails, dependent tasks won't execute, preventing data inconsistencies.
- **Resource Optimization** – Ensures clusters are only used when required, reducing costs.

3. How to Set Up a Predecessor Task

In Databricks, you can define predecessor tasks while creating or editing a multi-task job.

Steps to Configure a Predecessor Task in Databricks

1. **Navigate to Databricks Workflows**
   - Open the Databricks UI and go to **Workflows**.
   - Click **Create Job** or edit an existing job.
2. **Add a New Task**
   - Click **Add Task** and define its properties (task name, type, compute cluster, etc.).
3. **Define Task Dependencies (Predecessor)**
   - In the **Task Dependencies** section, choose the **predecessor task** that must complete before this task runs.
   - You can define multiple predecessor tasks for complex workflows.
4. **Set Task Execution Conditions**
   - Configure the task to execute **only if its predecessors succeed** or allow execution under specific failure conditions.
5. **Save and Run the Job**
   - Click **Save** and trigger the job to verify dependency execution.

4. Business-Oriented Example: Data Processing Pipeline Scenario: Financial Data Processing Workflow

A financial institution processes transaction records daily and must ensure data integrity before generating reports. The following **multi-task job workflow** ensures proper execution:

Task Name	Description	Predecessor Task

Ingest Transactions	Fetch raw transaction data from sources	None
Validate Data	Perform data quality checks	Ingest Transactions
Transform Data	Cleanse and format transaction records	Validate Data
Aggregate Metrics	Compute total revenue, fraud detection	Transform Data
Generate Reports	Create financial reports for review	Aggregate Metrics

Key Benefits Observed

- **Ensured Data Integrity** – Data validation runs before aggregation, preventing incorrect reports.
- **Efficient Execution** – Dependencies allow tasks to execute only when necessary, reducing delays.
- **Optimized Resource Usage** – Each task runs on an appropriate cluster, reducing unnecessary compute costs.
- **Task Dependency Management** – Setting up and modifying task execution order.
- **Failure Handling Strategies** – Configuring tasks to retry or stop based on predecessor task failures.
- **Parallel s Sequential Execution** – Understanding when tasks should run concurrently vs. sequentially.
- **Job Scheduling Best Practices** – Optimizing job dependencies for better performance and cost-efficiency.

4. Identify a scenario in which a predecessor task should be set up.

Scenario Where a Predecessor Task Should Be Set Up in Databricks Jobs

1. Introduction

A **predecessor task** ensures that dependent tasks execute in the correct order, preventing errors, improving efficiency, and maintaining data integrity.

2. Scenario: ETL Pipeline for Customer

Data Processing Business Use Case:

A **retail company** collects customer transactions, demographics, and website

interactions. The data engineering team needs to create an **ETL (Extract, Transform, Load) pipeline** to process and store this data for analytics.

Challenges Without Predecessor Tasks:

- Data may be transformed before extraction is complete, leading to missing or inconsistent records.
- Reports may be generated with incomplete data if aggregation starts before all data is available.
- Compute resources may be inefficiently used if independent tasks run without dependencies.

Recommended Workflow with Predecessor Tasks

Task Name	Description	Predecessor Task
Extract Data	Retrieve raw data from various sources	None
Validate Data	Perform data quality checks on raw data	Extract Data
Transform Data	Cleanse, standardize, and format data	Validate Data
Load to Warehouse	Store processed data into a data warehouse	Transform Data
Generate Reports	Create business intelligence reports	Load to Warehouse

3. Key Benefits of Setting Up a Predecessor Task

- **Ensures Data Completeness** – Prevents reports from using incomplete or unverified data.
- **Reduces Processing Errors** – Data transformation and loading occur only after validation.
- **Optimizes Resource Allocation** – Compute resources are allocated based on execution order, reducing idle time.
- **Enhances Debugging s Monitoring** – Easier to track errors and retry specific steps without re-running the entire workflow.
- **Task Dependency Configuration** – Setting up a logical flow of execution.
- **Workflow Optimization** – Avoiding unnecessary delays while maintaining data integrity.
- **Error Handling Strategies** – Configuring retries and alerts based on predecessor failures.

- **Parallel vs. Sequential Execution** – Understanding when tasks should run sequentially vs. in parallel.

5. Review a task's execution history.

Reviewing a Task's Execution History in Databricks Jobs

1. Introduction

Monitoring and reviewing a task's execution history in Databricks Jobs is essential for identifying failures, optimizing performance, and ensuring workflow efficiency. Regularly reviewing a task's execution history in Databricks Jobs helps ensure operational efficiency, enhances debugging, and optimizes resource allocation.

2. Why Review a Task's Execution History?

- **Identify Failures s Debug Issues** – Quickly find errors and determine their causes.
- **Optimize Performance** – Analyze execution times to enhance efficiency.
- **Ensure Data Integrity** – Verify task completion and prevent data inconsistencies.
- **Monitor Resource Utilization** – Track compute usage to optimize costs.

3. How to Review a Task's Execution History in Databricks Steps to Access Execution History

1. **Navigate to Databricks Workflows**
    - Open the Databricks UI and go to **Workflows**.
    - Select the **Job** you want to inspect.
2. **Open Job Run Details**
    - Click on the specific **Job Run ID** to view execution details.
3. **View Task Execution History**
    - Under the **Run Details** tab, check individual task execution status.
    - Examine logs, timestamps, and retry attempts.

4. **Analyze Logs and Metrics**
    - Access **Spark UI logs** for deeper analysis.
    - Review error messages and stack traces if a failure occurred.
5. **Download Logs for Further Debugging**
    - Click **Download Logs** for offline analysis if needed.

4. Business-Oriented Example: Monitoring an ETL Pipeline Scenario: Financial Transaction Data Processing

A **banking institution** processes millions of daily transactions using a Databricks ETL pipeline. The data engineering team needs to review execution history to:

- Detect failed transactions.
- Analyze processing times to meet SLA requirements.
- Optimize cluster allocation based on task execution duration.

Execution History Analysis

Task Name	Execution Time	Status	Logs s Observations
Extract Data	5 min	Success	No issues detected
Validate Data	3 min	Failed	Schema mismatch error
Transform Data	-	Skipped	Not executed due to failure
Load to Warehouse	-	Skipped	Not executed due to failure

Key Takeaways

- The **Validate Data** task failed due to a schema mismatch, preventing downstream tasks from running.
- Reviewing execution history helped **identify the root cause quickly**.
- The team can **adjust schema validation logic** and rerun the failed task instead of re- executing the entire job.
- **Monitoring s Debugging** – Learn how to access and analyze execution history.

- **Error Handling** – Implement retry strategies based on execution failures.
- **Job Performance Tuning** – Use execution history insights to optimize workflows.
- **Logging s Observability** – Understand how to leverage logs for troubleshooting.

CRON Scheduling, Debugging Failed Tasks, and Setting Up Retry Policies

1. **Identifying CRON as a Scheduling Opportunity What is CRON?**

CRON is a time-based job scheduler in Unix-like operating systems that automates script execution at scheduled times or intervals. It is commonly used for system maintenance, backups, log rotations, and other repetitive tasks.

Key Features of CRON:

- Automates repetitive tasks.
- Runs commands or scripts at predefined times and intervals.
- Uses a simple syntax to define execution schedules.
- Provides logging capabilities to track task execution.

CRON Syntax:

A CRON job follows the syntax:

* * * * * command_to_be_executed

Day of the week (0 - 7) [0 and 7 represent Sunday]

Month (1 - 12)

Day of the month

(1 - 31) Hour (0 -

23)

Minute (0 - 59)

Examples of CRON Jobs:

1. Run a script every day at midnight:
2. 0 0 * * * /path/to/script.sh
3. Run a backup every Sunday at 2 AM:

4. 0 2 * * 0 /path/to/backup.sh
5. Clear cache every 6 hours:
6. 0 */6 * * * /path/to/cache_cleanup.sh

Benefits of Using CRON for Scheduling:

- Efficient execution of periodic tasks.
- Reduces manual intervention.
- Improves system reliability and performance.
- Enhances workflow automation.

2. **Debugging a Failed**

Task Common

Causes of Task

Failure:

- Incorrect CRON syntax.
- Script permission issues.
- Environment variables not set correctly.
- Dependency failures (e.g., missing files, databases not accessible).
- Insufficient system resources.

Steps to Debug a Failed CRON Job:

1. **Check the CRON Log:**

Most Linux distributions log CRON executions in /var/log/syslog or /var/log/cron.log. grep CRON /var/log/syslog

2. **Check the Mail Log:**

CRON jobs often send output to the system mail of the user. Run:

mail

3. **Redirect Output to a Log File:**

Modify the CRON job to capture output:

* * * * * /path/to/script.sh >> /path/to/logfile.log 2>&1

4. **Test the Script Manually:**

Try running the script outside CRON:

/bin/bash /path/to/script.sh

5. **Check File and Execution Permissions:**

Ensure the script has executable permissions:

chmod +x /path/to/script.sh

6. **Verify the User Environment:**

Environment variables may not be set in CRON. Use absolute paths and define variables explicitly in the script. PATH=/usr/local/bin:/usr/bin:/bin

3. **Setting Up a Retry Policy in Case of Failure Why Use a Retry Policy?**

- Ensures critical tasks are completed even in case of temporary failures.
- Reduces manual intervention in case of transient issues.
- Improves system resilience and reliability.

Methods to Implement Retry Policies:

1. **Using CRON with Built-in Retries:**

Modify the CRON job to retry execution using a loop:

* * * * * for i in {1..3}; do /path/to/script.sh && break || sleep 60; done This will retry up to 3 times with a 60-second delay between retries.

2. **Using a Wrapper Script for Retries:** Create a wrapper script that retries execution: #!/bin/bash

MAX_RETRIES=3

```
RETRY_COUNT=0
while [ $RETRY_COUNT -lt $MAX_RETRIES ]; do
  /path/to/script.sh      &&      break
  RETRY_COUNT=$((RETRY_C
  OUNT+1))
  s
le
ep
6
0
d
o
ne
```

Schedule this script in CRON instead of the original script.

3. **Using Systemd Timers for Enhanced Retry Mechanisms:**

Systemd timers provide an alternative to CRON with

better logging and retry policies. [Unit]

Description=Run script with retries

```
[Service]
ExecStart=/p
ath/to/script.s
h
Restart=on-
failure
```

RestartSec=6

0s

[Install]

WantedBy=mul

ti-user.target

Enable the

service and start

it:

systemctl enable

myscript.service

systemctl start

myscript.service

Best Practices for Retry Policies:

- Use exponential backoff to avoid overloading the system.
- Log each retry attempt for easier debugging.
- Implement alerting mechanisms for persistent failures.
- Set a maximum retry limit to avoid infinite loops.

❖ **Compare and contrast metastores and catalogs.**

- **Metastore**:
    - A metastore is the top-level container for metadata in Databricks Unity Catalog.
    - It stores metadata such as tables, views, and permissions for data assets.
    - Each metastore is tied to a specific AWS account and region.
    - A single metastore can be shared across multiple Databricks workspaces within the same AWS account and region.

- It acts as a centralized metadata repository for managing data assets.

- **Catalog**:
  - A catalog is a collection of schemas (databases) within a metastore.
  - It is a logical grouping of databases and tables, often used to organize data by team, project, or environment (e.g., dev, prod).
  - Catalogs are part of the three-level namespace in Unity Catalog: **catalog.schema.table**.
  - Unlike a metastore, a catalog is not tied to a specific AWS account or region but is contained within a metastore.

Key Differences:

- A metastore is a higher-level container that spans multiple workspaces, while a catalog is a lower-level container within a metastore.
- Metastores are tied to AWS accounts and regions, whereas catalogs are not.
- Metastores store metadata for all data assets, while catalogs organize data into logical groups.

❖ **Identify Unity Catalog securables.**

Securables in Unity Catalog are entities that can have permissions applied to them. These Include:

- **Metastore**: The top-level container for metadata.
- **Catalog**: A collection of schemas (databases).
- **Schema (Database)**: A collection of tables and views.
- **Table**: A structured data entity.
- **View**: A virtual table defined by a query.
- **Column**: A specific field within a table or view.
- **Function**: A user-defined function (UDF) that can be used in

queries.

Permissions can be granted or revoked on these securables to control access to data and metadata.

- **Define a service principal.**

A **service principal** is an identity used by an application or service to access Databricks resources programmatically. It is similar to a user account but is intended for automated workflows, scripts, or integrations rather than human users. Key characteristics include:

- It is created and managed in Databricks.
- It can be assigned roles and permissions, just like a user.
- It is often used for CI/CD pipelines, ETL processes, or other automated tasks.
- Service principals are authenticated using tokens or secrets, allowing secure access to Databricks APIs and resources.

- **Identify one of the four areas of data governance.**

Data Cataloging

1. **Centralized Metadata Repository**:
   - A data catalog acts as a single source of truth for metadata about an organization's data assets.
   - It stores information about data such as its format, structure, location, and usage.
2. **Improved Data Discovery**:
   - Stakeholders can quickly search and find the data they need.
   - Reduces time spent locating relevant data for analysis or decision-making.
3. **Enhanced Understanding of Data**:
   - Provides context about data, such as its meaning, origin, and relationships.

- Helps users understand how data can be used effectively.

4. **Supports Data Governance**:
   - Enables better management of data assets by tracking ownership, usage, and access controls.
   - Ensures compliance with data policies and regulations.
5. **Boosts Collaboration**:
   - Teams can easily share and access data, reducing silos and improving teamwork.
   - Promotes consistency and reduces redundancy in data usage.
6. **Searchable Index**:
   - Acts like a search engine for data, making it easy to locate specific datasets or information.
   - Provides semantic value by organizing unstructured data into a usable format.
7. **Access Control and Auditing**:
   - Ensures proper access controls are in place to protect sensitive data.
   - Tracks data usage and retrieval for auditing purposes.
8. **Improves Data Management**:
   - Helps organizations streamline data-related activities like analytics, reporting, and governance.
   - Reduces inefficiencies and enhances the overall quality of data operations.

Other Key Elements of Data Governance:

- Data Quality
- Data Classification
- Data Security
- Auditing Data Entitlements and Access
- Data Lineage
- Data Discovery
- Data Sharing and Collaboration

Databricks Unity Catalog: Security Modes, Cluster s Warehouse Creation, and Querying the Namespace

1. Cluster Security Modes Compatible with Unity Catalog in Databricks

Databricks Unity Catalog provides a centralized governance layer for managing and securing data across multiple workspaces. To ensure compliance with governance and security policies, Unity Catalog requires specific cluster security modes that enforce
access controls and user identity management. This document explores the security modes that are compatible with Unity Catalog and explains their features, benefits, and considerations.

Understanding Cluster Security Modes

Databricks clusters operate in different security modes that define how users and workloads interact with data. The selection of an appropriate security mode is crucial for ensuring that Unity Catalog's governance framework functions effectively. The
following security modes are compatible with Unity Catalog:

1. Single User Access Mode

Description: Single User Access Mode is a highly secure environment where a single
user owns and operates the cluster. Only the cluster owner can execute commands and access data, ensuring strict data isolation and control.

Key Features:

- Restricted to a single user for execution.
- Enforces fine-grained access controls via Unity Catalog.
- Supports identity passthrough, ensuring that the user's identity is used for authorization.
- Ideal for scenarios where sensitive data is accessed, preventing unintended data sharing.

Use Cases:

- Running personal workloads that require data isolation.
- Developing and testing sensitive queries before deploying them to a shared environment.
- Enforcing strict access policies for regulatory compliance.

Considerations:

- Limits collaboration as only one user can run jobs and queries.
- Not suitable for workloads requiring multiple users on the same cluster.

2. Shared Access Mode

Description: Shared Access Mode allows multiple users to share a cluster while maintaining governance and security controls via Unity Catalog. This mode is designed for collaborative environments where multiple users need to access and analyze data simultaneously.

Key Features:

- Supports multiple users while enforcing Unity Catalog's security policies.
- Enforces role-based access controls (RBAC) to restrict user access.
- Provides centralized governance, ensuring that users can only access authorized data.
- Enables cost efficiency by allowing shared resource utilization.

Use Cases:

- Collaborative data analysis and exploration.
- Running production workloads with controlled data access.
- Managing workloads that require multiple contributors within a governed environment.

Considerations:

- Requires careful role and permission management to avoid unauthorized access.
- May introduce security risks if governance policies are not correctly enforced.

3. No Isolation Shared Mode (Not Compatible with Unity Catalog)

Description: No Isolation Shared Mode is a legacy security mode that lacks the governance and security controls required for Unity Catalog. This mode does not enforce identity-based access controls and should not be used when working with Unity Catalog-enabled clusters.

Key Features:

- Allows unrestricted access to users without governance

enforcement.

- Does not support Unity Catalog's access control and auditing features.
- Poses security risks by allowing users to access data without restriction.

Use Cases:

- Not recommended for Unity Catalog-enabled environments.
- Suitable for open data exploration where governance is not a concern.

Considerations:

- Does not provide data security or access control.
- Can lead to data governance and compliance violations.

Choosing the Right Security Mode for Unity Catalog

Selecting the appropriate security mode is essential for ensuring that Unity Catalog
functions correctly and enforces governance policies. The following guidelines can help in choosing the right security mode:

Security Mode	**Supports Unity Catalog?**	**Best Use Cases**
Single User	Yes	Personal workloads, sensitive data analysis
Shared Access	Yes	Collaborative analysis, production workloads
No Isolation Shared	No	Open data exploration (not recommended for UC)

2. Creating a Unity Catalog-Enabled All-Purpose Cluster

To work with Unity Catalog in Databricks, it is essential to create a UC-enabled all- purpose cluster. The cluster must be properly configured to support Unity Catalog's security and governance features.

Step 1: Navigate to the Compute Section

1. Log in to **Databricks**.

2. Click on **Compute** in the left navigation panel.
3. Click the **Create Cluster** button.

Step 2: Configure Cluster Settings

General Cluster Configuration:

- **Cluster Name**: Provide a meaningful name (e.g., UC_Enabled_Cluster).
- **Databricks Runtime Version**: Select a runtime version that supports Unity Catalog.
- **Worker Type s Number of Workers**: Configure based on workload requirements.
- **Driver Type**: Choose a driver instance suitable for job execution.

Set Cluster Mode s Security Mode:

- **Cluster Mode**: Select **Standard** (for general workloads) or **High Concurrency** (for SQL-heavy workloads).
- **Access Mode**: Choose **Shared** or **Single User** to enable Unity Catalog compatibility.
- **Security Mode**: Ensure Unity Catalog support by selecting a **Unity Catalog- compatible security mode**.

Step 3: Enable Unity Catalog Integration

1. **Attach Unity Catalog Metastore:**
    - Navigate to the **Advanced Options**.
    - Select **Enable Unity Catalog** and choose the appropriate **Metastore**.
2. **Enable Identity Passthrough (if needed):**
    - If your organization enforces identity-based access controls, enable **Credential Passthrough**.

Step 4: Set Permissions and Governance Policies

1. Assign the **cluster owner** and **users/groups** who can access it.
2. Ensure **RBAC permissions** align with Unity Catalog governance policies.
3. Configure **network policies** if required for data security.
4. Define **workspace-level permissions** to restrict unauthorized users from accessing sensitive data.

Step 5: Launch and Validate the Cluster

1. Click **Create Cluster**.
2. Wait for the cluster to start.
3. Validate **Unity Catalog connectivity** by running test queries on UC-managed tables.

Best Practices for Creating a UC-Enabled Cluster

- Always select a **Databricks Runtime that supports Unity Catalog**.
- Use **Shared Access Mode** for collaboration and **Single User Mode** for secure workloads.
- Assign **appropriate user roles** to control data access.
- Regularly **monitor cluster logs** to ensure compliance with security policies.
- Maintain an **audit trail** for all cluster activities to track data access and modifications.
- Optimize **cost efficiency** by choosing the right instance types and autoscaling settings.

By following these steps, users can successfully create a Unity Catalog-enabled cluster, ensuring secure and governed data access while leveraging Databricks' capabilities efficiently.

3. Creating a DBSQL Warehouse

A **Databricks SQL (DBSQL) Warehouse** is a compute resource optimized for running SQL queries on Databricks. It is specifically designed to provide efficient execution for data analysts and business

intelligence workloads.

Steps to Create a DBSQL Warehouse

1. **Navigate to SQL Warehouses**
    - In the Databricks workspace, go to the left panel and click **SQL Warehouses**.
2. **Click on "Create SQL Warehouse"**
    - Provide a meaningful name for the warehouse.
3. **Configure Settings**
    - Choose the appropriate instance type and size.
    - Configure auto-scaling options.
    - Set up the warehouse with necessary access control and governance settings.
4. **Enable Unity Catalog (if applicable)**
    - Attach the SQL warehouse to a Unity Catalog metastore.
5. **Set Permissions and Access Controls**
    - Define who can use the warehouse.
6. **Launch the Warehouse**
    - Click **Create** and wait for it to initialize.

By following these steps, users can successfully create and configure a DBSQL warehouse for running SQL workloads efficiently.

4. Identifying How to Query a Three-Layer Namespace

Unity Catalog follows a structured three-layer namespace format:

• Catalog: The highest level, representing a collection of schemas (e.g., finance catalog).

• Schema: A logical grouping of tables within a catalog (e.g., transactions schema inside finance).

• Table: The actual data storage unit within a schema (e.g., payments table in transactions).

Querying a Three-Layer Namespace

To query data in Unity Catalog, use the following syntax:

SELECT * FROM catalog_name.schema_name.table_name;

Example:

SELECT * FROM finance_db.sales_data.transactions;

This approach ensures that data is accessed in a well-structured manner, supporting governance and security policies.

Databricks Data Governance

Introduction

Data governance in Databricks is essential for ensuring secure, compliant, and efficient data management across an organization. It involves defining policies and controls for data access, security, and organization-wide consistency. Implementing best practices in data governance helps organizations protect sensitive information, maintain regulatory compliance, and optimize data workflows. This document outlines key governance practices, including access control, metastore colocation, service principals, and catalog segregation.

1. Implementing Data Object Access Control

Overview

Data Object Access Control in Databricks ensures that users and groups have appropriate permissions to access, modify, or administer data objects such as tables, views, and files. Databricks leverages Unity Catalog for

fine-grained access control across workspaces.

Implementation

- **Unity Catalog:** Provides a centralized governance model across multiple Databricks workspaces.
- **Permissions Management:** Use SQL GRANT and REVOKE statements to assign roles (e.g., SELECT, INSERT, MODIFY) to users or groups.
- **Attribute-Based Access Control (ABAC):** Enforce policies based on user attributes and metadata.
- **Row-Level & Column-Level Security:** Restrict data access at a more granular level.

Example Scenario

A retail company wants to restrict sales data access based on regional managers. GRANT SELECT ON TABLE sales_data TO `manager_east` WHERE region = 'East'; GRANT SELECT ON TABLE sales_data TO `manager_west` WHERE region = 'West';

2. Colocating Metastores with a Workspace

Overview

Colocating metastores with a workspace enhances performance and compliance. This practice ensures that metadata operations are executed within the same geographic region as the data.

Best Practices

- Deploy Unity Catalog in the same cloud region as Databricks workspaces.
- Ensure data compliance by aligning with regional regulations (e.g., GDPR, CCPA).
- Optimize performance by reducing network latency between the metastore and compute resources.

Example Scenario

A financial institution operates in the EU and must comply with GDPR. They place their metastore in the same AWS Europe region as their primary Databricks workspace.

aws s3 ls s3://eu-central-1-databricks-metastore

3. Using Service Principals for Connections

Overview

Service Principals are non-human identities used to authenticate applications, workflows, and services securely in Databricks.

Best Practices

- Use service principals for API and ETL pipeline authentication

instead of personal credentials.

- Assign minimal required permissions to service principals.
- Rotate credentials regularly to prevent unauthorized access.

Example Scenario

A data pipeline needs access to an Azure Data Lake storage account. A service principal is created and assigned permissions.

az ad sp create-for-rbac --name "databricks-pipeline" --role "Storage Blob Data Contributor"-

-

scopes/subscriptions/{subscriptionid}/resourceGroups/{re

source-

group}/providers/Microsoft.Storage/storageAccounts/{stor

age-account}

4. Segregating Business Units Across Catalogs

Overview

Segregating business units across catalogs ensures data isolation and security within an organization.

Best Practices

- Create separate catalogs for different business units (e.g., Finance, Marketing, HR).
- Assign access permissions based on business unit roles.
- Implement data lineage tracking for auditability.

Example Scenario

A company wants to segregate Finance and Marketing data into separate catalogs. CREATE CATALOG finance;

CREATE CATALOG marketing;

GRANT USAGE ON CATALOG finance TO `finance_team`; GRANT USAGE ON CATALOG marketing TO `marketing_team`;

Conclusion

Implementing robust data governance in Databricks ensures security, compliance, and efficiency in data access and management. Following best practices for access control, metastore colocation, service principals, and catalog segregation helps organizations build scalable and secure data solutions.

Databricks Documentation

- Jovin Joy Arakkal

1) Identify how the count_if function and the count where x is null function can be used?

Count_if function

The count_if function counts the number of rows that satisfy a specific condition that is provided and returns the count or NULL if no records satisfy the specified condition. Unlike count(*), which counts all rows, or count(column), which counts non-NULL values in a column, count_if filters rows dynamically during aggregation. This is mainly used in situations like real time data analysis where filtering and counting specific subsets of data are essential.

Its syntax is:

```
df.select(count_if(col('c2') % 2 == 0))
```

Count Where x is NULL function

The count where x is NULL function is used to count the number of records having NULL value in a column as a conditional expression. Here x stands for the column name. This is mainly used to identify NULL values in particular columns as if left unchecked, NULL values can skew analytics outcomes. It can also be used to identify gaps in data collection. Removing these can increase efficiency, enhance data authenticity, scalability and performance.

Its syntax is:

```
df.select([count(when(col(c).contains('None')
| \
        col(c).contains('NU
        LL') | \ (col(c) == '' )
        | \ col(c).isNull() | \
        isnan(c), c
       )).alias(c)
    for c in df.columns])
```

2) Identify that an alert can be sent via email

Alerting is a critical component of workflows that serve as an early warning system. It helps by enabling organizations to detect, diagnose and resolve issues before they escalate into disruptions that have high cost. There are many methods to send an alert by email. They are:

Using Python and SMTP

After setting up the credentials successfully, store SMTP credentials using Databricks secrets.

```
import smtplib
```

```
from email.message import EmailMessage def send_email(subject, body, to):
  msg = EmailMessage()
  msg.set_content(body)
  msg['Subject'] = subject
  msg['From'] = "your-email@domain.com"
  msg['To'] = to
with smtplib.SMTP_SSL('smtp.gmail.com', 465) as server:
    server.login(dbutils.secrets.get("scope", "email_user"),
          dbutils.secrets.get("scope", "email_pass"))
    server.send_message(msg)
```

Third-Party Email Services

Some third-party email services like SendGrid, Mailgun also provide scalable and reliable email delivery for alerts. In this case we need to create accounts for the respective service and generate a key which we can use for our alert mails.

Cloud-Native Email Solutions

We can leverage cloud provider services like AWS SES, Azure Logic Apps etc to decouple email logic from Databricks

Native Databricks Features

In the Databricks Jobs UI, we can configure email alerts for job status. However, these are basic alerts only. We cannot send

custom messages or dynamic content. We can also create alerts in Databricks SQL to monitor query results

3) Create an alert in the case of a failed task

Create Databricks secret which contains username and password.

```
import smtplib
from email.message import EmailMessage
def send_failure_alert(task_name, error_message, recipient):
  try:
    sender =
    dbutils.secrets.get(scope="email_alerts",
    key="gmail_user") password =
    dbutils.secrets.get(scope="email_alerts",
    key="gmail_pass") msg = EmailMessage()
    msg['From'] = "your-
    email@domain.com"
    msg['To'] = to
    msg['Subject'] = f"ALERT: Task
    '{task_name}' Failed"
    msg.set_content(f"""
    Task Failure Details:
    - Task: {task_name}
    - Error: {error_message}
    - Timestamp:
    {datetime.now().strftime("%Y-%m-%d
    %H:%M:%S")} Action Required: Check
```

```
Databricks Job logs immediately.
""")
with smtplib.SMTP_SSL('smtp.gmail.com', 465) as
server: server.login(dbutils.secrets.get("scope",
"email_user"),
    dbutils.secrets.get("scope", "email_pass"))
server.send_message(msg)
print("Alert email sent
successfully.") except
Exception as e:
print(f"Failed to send alert: {str(e)}")
```

We can integrate the alert into our task as part of code or configure to run automatically on failure of task in Databricks Job UI or webhooks etc or we can further automate this using Databricks REST APIs or Azure Logic Apps or AWS Lambda to trigger notifications based on the status of your Databricks workflows.

Deduplicate rows from an existing Delta Lake table

In this we can deduplicate or remove duplicate records from a delta lake by creating a new field called row number based on some unique fields and then select only the rows which have row number as one.

Using Row_Num

First, we load the Delta Table into a DataFrame. Then we perform Deduplication based on the columns of interest. And finally, we write back to Delta table with deduplicated table

```
from pyspark.sql.window import Window
from pyspark.sql.functions import
row_number delta_table_path =
```

```
"/path/to/your/delta-table"
dedup_columns = ["order_id",
"customer_id"]
window_spec =
Window.partitionBy(dedup_columns).orderBy("time
stamp") df =
spark.read.format("delta").load(delta_table_path
)
dedup_df = df.withColumn("row_num",
row_number().over(window_spec))
dedup_df.createOrReplaceTempView("temp_dedup_view")
spark.sql(f"""
  DELETE FROM {table_name}
  WHERE (order_id,
    customer_id, timestamp) IN (
    SELECT order_id,
    customer_id, timestamp
    FROM temp_dedup_view
    WHERE row_num > 1)
""")
deduplicated_df.write.format("delta").mode("overwrite").save(delta
_table_path)
```

Using MERGE

Another method which is optimal if we are dealing with incrementally loaded tables is to use MERGE for incremental upsert which updates record when there is match and inserts when there is no match

```
from delta.tables import DeltaTable
delta_table = DeltaTable.forPath(spark, delta_table_path)
delta_table.alias("target").merge(
  deduplicated_df.alias("so
  urce"), "target.id =
  source.id"
).whenMatchedUpdate(
  condition="target.timestamp <
  source.timestamp",
  set={"id": "source.id", "name": "source.name", "timestamp":
  "source.timestamp"}
```

```
).whenNotMatchedInsert(
  values={"id": "source.id", "name": "source.name", "timestamp":
  "source.timestamp"}
).execute()
```